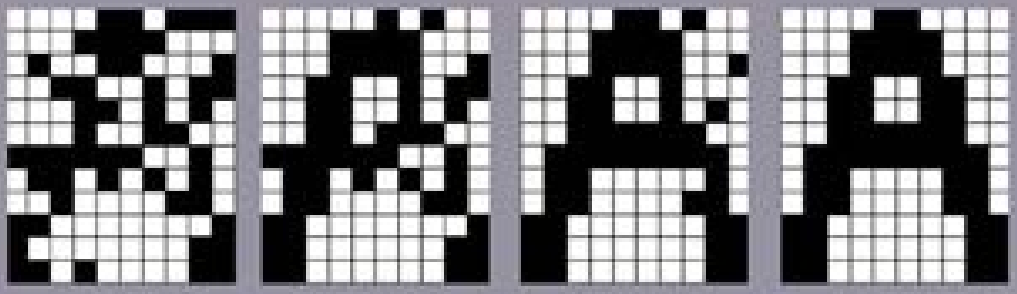
VISIT…

TEXTS IN COMPUTER SCIENCE

Fundamentals of the New Artificial Intelligence

Neural, Evolutionary, Fuzzy and More



Toshinori Munakata

SECOND EDITION

Springer

TEXTS IN COMPUTER SCIENCE

Editors
David Gries
Fred B. Schneider

TEXTS IN COMPUTER SCIENCE

Apt and Olderog, Verification of Sequential and Concurrent Programs, Second Edition

Alagar and Periyasamy, Specification of Software Systems

Back and von Wright, Refinement Calculus: A Systematic Introduction

Beidler, Data Structures and Algorithms: An Object-Oriented Approach Using Ada 95

Bergin, Data Structures Programming: With the Standard Template Library in C++

Brooks, C Programming: The Essentials for Engineers and Scientists

Brooks, Problem Solving with Fortran 90: For Scientists and Engineers

Dandamudi, Fundamentals of Computer Organization and Design

Dandamudi, Introduction to Assembly Language Programming: For Pentium and RISC Processors, Second Edition

Dandamudi, Introduction to Assembly Language Programming: From 8086 to Pentium Processors

Fitting, First-Order Logic and Automated Theorem Proving, Second Edition

Grillmeyer, Exploring Computer Science with Scheme

Homer and Selman, Computability and Complexity Theory

Immerman, Descriptive Complexity

Jalote, An Integrated Approach to Software Engineering, Third Edition

(continued after index)

Toshinori Munakata

Fundamentals of the New Artificial Intelligence

Neural, Evolutionary, Fuzzy and More

Second Edition

Springer

Toshinori Munakata
Computer and Information Science Department
Cleveland State University
Cleveland, OH 44115
USA
t.munakata@csuohio.edu

ISBN: 978-1-84628-838-8 e-ISBN: 978-1-84628-839-5
DOI: 10.1007/978-1-84628-839-5

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2007929732

© Springer-Verlag London Limited 2008
Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act of 1988, this publication may only be reproduced, stored or transmitted, in any form or by any means, with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licences issued by the Copyright Licensing Agency. Enquiries concerning reproduction outside those terms should be sent to the publishers.
The use of registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant laws and regulations and therefore free for general use.
The publisher makes no representation, express or implied, with regard to the accuracy of the information contained in this book and cannot accept any legal responsibility or liability for any errors or omissions that may be made.

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

Springer Science+Business Media
springer.com

Preface

This book was originally titled "Fundamentals of the New Artificial Intelligence: Beyond Traditional Paradigms." I have changed the subtitle to better represent the contents of the book. The basic philosophy of the original version has been kept in the new edition. That is, the book covers the most essential and widely employed material in each area, particularly the material important for real-world applications. Our goal is not to cover every latest progress in the fields, nor to discuss every detail of various techniques that have been developed. New sections/subsections added in this edition are: Simulated Annealing (Section 3.7), Boltzmann Machines (Section 3.8) and Extended Fuzzy if-then Rules Tables (Sub-section 5.5.3). Also, numerous changes and typographical corrections have been made throughout the manuscript. The Preface to the first edition follows.

General scope of the book

Artificial intelligence (AI) as a field has undergone rapid growth in diversification and practicality. For the past few decades, the repertoire of AI techniques has evolved and expanded. Scores of newer fields have been added to the traditional symbolic AI. Symbolic AI covers areas such as knowledge-based systems, logical reasoning, symbolic machine learning, search techniques, and natural language processing. The newer fields include neural networks, genetic algorithms or evolutionary computing, fuzzy systems, rough set theory, and chaotic systems.

The traditional symbolic AI has been taught as the standard AI course, and there are many books that deal with this aspect. The topics in the newer areas are often taught individually as special courses, that is, one course for neural networks, another course for fuzzy systems, and so on. Given the importance of these fields together with the time constraints in most undergraduate and graduate computer science curricula, a single book covering the areas at an advanced level is desirable. This book is an answer to that need.

Specific features and target audience

The book covers the most essential and widely employed material in each area, at a level appropriate for upper undergraduate and graduate students. Fundamentals of both theoretical and practical aspects are discussed in an easily understandable

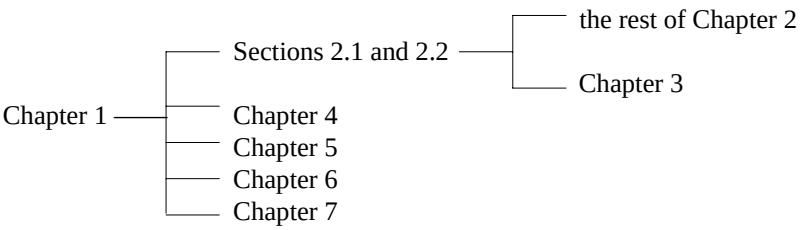
fashion. Concise yet clear description of the technical substance, rather than journalistic fairy tale, is the major focus of this book. Other non-technical information, such as the history of each area, is kept brief. Also, lists of references and their citations are kept minimal.

The book may be used as a one-semester or one-quarter textbook for majors in computer science, artificial intelligence, and other related disciplines, including electrical, mechanical and industrial engineering, psychology, linguistics, and medicine. The instructor may add supplementary material from abundant resources, or the book itself can also be used as a supplement for other AI courses.

The primary target audience is seniors and first- or second-year graduates. The book is also a valuable reference for researchers in many disciplines, such as computer science, engineering, the social sciences, management, finance, education, medicine, and agriculture.

How to read the book

Each chapter is designed to be as independent as possible of the others. This is because of the independent nature of the subjects covered in the book. The objective here is to provide an easy and fast acquaintance with any of the topics. Therefore, after glancing over the brief Chapter 1, Introduction, the reader can *start from any chapter*, also proceeding through the remaining chapters in any order depending on the reader's interests. An exception to this is that Sections 2.1 and 2.2 should precede Chapter 3. In diagram form, the required sequence can be depicted as follows.



The relationship among topics in different chapters is typically discussed close to the end of each chapter, whenever appropriate.

The book can be read without writing programs, but coding and experimentation on a computer is essential for complete understanding these subjects. Running so-called canned programs or software packages does not provide the target comprehension level intended for the majority of readers of this book.

Prerequisites

Prerequisites in mathematics. College mathematics at freshman (or possibly at sophomore) level are required as follows:

Chapters 2 and 3 Neural Networks:	Calculus, especially partial differentiation, concept of vectors and matrices, and elementary probability.

Chapter 4 Genetic algorithms:	Discrete probability.
Chapter 5 Fuzzy Systems:	Sets and relations, logic, concept of vectors and matrices, and integral calculus.
Chapter 6 Rough Sets:	Sets and relations. Discrete probability.
Chapter 7 Chaos:	Concept of recurrence and ordinary differential equations, and vectors.

Highlights of necessary mathematics are often discussed very briefly before the subject material. Instructors may further augment the basics if students are unprepared. Occasionally some basic mathematics elements are repeated briefly in relevant chapters for an easy reference and to keep each chapter independent as possible.

Prerequisites in computer science. Introductory programming in a conventional high-level language (such as C or Java) and data structures. Knowledge of a symbolic AI language, such as Lisp or Prolog, is not required.

Toshinori Munakata

Contents

Preface ..... **v**

1 Introduction **1**
1.1 An Overview of the Field of Artificial Intelligence 1
1.2 An Overview of the Areas Covered in this Book 3

2 Neural Networks: Fundamentals and the Backpropagation Model **7**
2.1 What is a Neural Network? 7
2.2 A Neuron 7
2.3 Basic Idea of the Backpropagation Model 8
2.4 Details of the Backpropagation Mode 15
2.5 A Cookbook Recipe to Implement the Backpropagation Model 22
2.6 Additional Technical Remarks on the Backpropagation Model 24
2.7 Simple Perceptrons 28
2.8 Applications of the Backpropagation Model 31
2.9 General Remarks on Neural Networks 33

3 Neural Networks: Other Models **37**
3.1 Prelude 37
3.2 Associative Memory 40
3.3 Hopfield Networks 41
3.4 The Hopfield-Tank Model for Optimization Problems: The Basics 46
3.4.1 One-Dimensional Layout 46
3.4.2 Two-Dimensional Layout 48
3.5 The Hopfield-Tank Model for Optimization Problems: Applications 49
3.5.1 The N-Queen Problem 49
3.5.2 A General Guideline to Apply the Hopfield-Tank Model to Optimization Problems 54
3.5.3 Traveling Salesman Problem (TSP) 55
3.6 The Kohonen Model 58
3.7 Simulated Annealing 63

3.8 Boltzmann Machines 69
3.8.1 An Overview 69
3.8.2 Unsupervised Learning by the Boltzmann Machine: The Basics Architecture 70
3.8.3 Unsupervised Learning by the Boltzmann Machine: Algorithms 76
3.8.4 Appendix. Derivation of Delta-Weights 81

4 Genetic Algorithms and Evolutionary Computing 85
4.1 What are Genetic Algorithms and Evolutionary Computing? 85
4.2 Fundamentals of Genetic Algorithms 87
4.3 A Simple Illustration of Genetic Algorithms 90
4.4 A Machine Learning Example: Input-to-Output Mapping 95
4.5 A Hard Optimization Example: the Traveling Salesman Problem (TSP) 102
4.6 Schemata 108
4.6.1 Changes of Schemata Over Generations 109
4.6.2 Example of Schema Processing 113
4.7 Genetic Programming 116
4.8 Additional Remarks 118

5 Fuzzy Systems 121
5.1 Introduction 121
5.2 Fundamentals of Fuzzy Sets 123
5.2.1 What is a Fuzzy Set? 123
5.2.2 Basic Fuzzy Set Relations 125
5.2.3 Basic Fuzzy Set Operations and Their Properties 126
5.2.4 Operations Unique to Fuzzy Sets 128
5.3 Fuzzy Relations 130
5.3.1 Ordinary (Nonfuzzy) Relations 130
5.3.2 Fuzzy Relations Defined on Ordinary Sets 133
5.3.3 Fuzzy Relations Derived from Fuzzy Sets 138
5.4 Fuzzy Logic 138
5.4.1 Ordinary Set Theory and Ordinary Logic 138
5.4.2 Fuzzy Logic Fundamentals 139
5.5 Fuzzy Control 143
5.5.1 Fuzzy Control Basics 143
5.5.2 Case Study: Controlling Temperature with a Variable Heat Source 150
5.5.3 Extended Fuzzy if-then Rules Tables 152
5.5.4 A Note on Fuzzy Control Expert Systems 155
5.6 Hybrid Systems 156
5.7 Fundamental Issues 157
5.8 Additional Remarks 158

6 Rough Sets 162
6.1 Introduction 162
6.2 Review of Ordinary Sets and Relations 165

6.3 Information Tables and Attributes 167
6.4 Approximation Spaces 170
6.5 Knowledge Representation Systems 176
6.6 More on the Basics of Rough Sets 180
6.7 Additional Remarks 188
6.8 Case Study and Comparisons with Other Techniques 191
6.8.1 Rough Sets Applied to the Case Study 192
6.8.2 ID3 Approach and the Case Study 195
6.8.3 Comparisons with Other Techniques 202

7 Chaos 206
7.1 What is Chaos? 206
7.2 Representing Dynamical Systems 210
7.2.1 Discrete dynamical systems 210
7.2.2 Continuous dynamical systems 212
7.3 State and Phase Spaces 218
7.3.1 Trajectory, Orbit and Flow 218
7.3.2 Cobwebs 221
7.4 Equilibrium Solutions and Stability 222
7.5 Attractors 227
7.5.1 Fixed-point attractors 228
7.5.2 Periodic attractors 228
7.5.3 Quasi-periodic attractors 230
7.5.4 Chaotic attractors 233
7.6 Bifurcations 234
7.7 Fractals 238
7.8 Applications of Chaos 242

Index 247

1 Introduction

1.1 An Overview of the Field of Artificial Intelligence

What is artificial intelligence?

The Industrial Revolution, which started in England around 1760, has replaced human muscle power with the machine. Artificial intelligence (AI) aims at replacing human intelligence with the machine. The work on artificial intelligence started in the early 1950s, and the term itself was coined in 1956.

There is no standard definition of exactly what artificial intelligence is. If you ask five computing professionals to define "AI", you are likely to get five different answers. The *Webster's New World College Dictionary, Third Edition* describes AI as "the capability of computers or programs to operate in ways to mimic human thought processes, such as reasoning and learning." This definition is an orthodox one, but the field of AI has been extended to cover a wider spectrum of subfields. AI can be more broadly defined as "the study of making computers do things that the human needs intelligence to do." This extended definition not only includes the first, mimicking human thought processes, but also covers the technologies that make the computer achieve intelligent tasks even if they do not necessarily simulate human thought processes.

But what is intelligent computation? This may be characterized by considering the types of computations that do not seem to require intelligence. Such problems may represent the complement of AI in the universe of computer science. For example, purely numeric computations, such as adding and multiplying numbers with incredible speed, are not AI. The category of pure numeric computations includes engineering problems such as solving a system of linear equations, numeric differentiation and integration, statistical analysis, and so on. Similarly, pure data recording and information retrieval are not AI. This second category of non-AI processing includes most business data and file processing, simple word processing, and non-intelligent databases.

After seeing examples of the complement of AI, i.e., nonintelligent computation, we are back to the original question: what is intelligent computation? One common characterization of intelligent computation is based on the *appearance* of the problems to be solved. For example, a computer adding 2 + 2 and giving 4 is not

intelligent; a computer performing symbolic integration of $\sin^2 x\, e^{-x}$ is intelligent. Classes of problems requiring intelligence include inference based on knowledge, reasoning with uncertain or incomplete information, various forms of perception and learning, and applications to problems such as control, prediction, classification, and optimization.

A second characterization of intelligent computation is based on the *underlying mechanism* for biological processes used to arrive at a solution. The primary examples of this category are neural networks and genetic algorithms. This view of AI is important even if such techniques are used to compute things that do not otherwise appear intelligent.

Recent trends in AI

AI as a field has undergone rapid growth in diversification and practicality. From around the mid-1980s, the repertoire of AI techniques has evolved and expanded. Scores of newer fields have recently been added to the traditional domains of practical AI. Although much practical AI is still best characterized as advanced computing rather than "intelligence," applications in everyday commercial and industrial settings have grown, especially since 1990. Additionally, AI has exhibited a growing influence on other computer science areas such as databases, software engineering, distributed computing, computer graphics, user interfaces, and simulation.

Different categories of AI

There are two fundamentally different major approaches in the field of AI. One is often termed traditional *symbolic AI*, which has been historically dominant. It is characterized by a high level of abstraction and a macroscopic view. Classical psychology operates at a similar level. Knowledge engineering systems and logic programming fall in this category. Symbolic AI covers areas such as knowledge based systems, logical reasoning, symbolic machine learning, search techniques, and natural language processing.

The second approach is based on low level, microscopic biological models, similar to the emphasis of physiology or genetics. Neural networks and genetic algorithms are the prime examples of this latter approach. These biological models do not necessarily resemble their original biological counterparts. However, they are evolving areas from which many people expect significant practical applications in the future.

In addition to the two major categories mentioned above, there are relatively new AI techniques which include fuzzy systems, rough set theory, and chaotic systems or chaos for short. Fuzzy systems and rough set theory can be employed for symbolic as well as numeric applications, often dealing with incomplete or imprecise data. These nontraditional AI areas - neural networks, genetic algorithms or evolutionary computing, fuzzy systems, rough set theory, and chaos - are the focus of this book.

1.2 An Overview of the Areas Covered in this Book

In this book, five areas are covered: neural networks, genetic algorithms, fuzzy systems, rough sets, and chaos. Very brief descriptions for the major concepts of these five areas are as follows:

Neural networks	Computational models of the brain. Artificial neurons are interconnected by edges, forming a neural network. Similar to the brain, the network receives input, internal processes take place such as activations of the neurons, and the network yields output.
Genetic algorithms:	Computational models of genetics and evolution. The three basic ingredients are selection of solutions based on their fitness, reproduction of genes, and occasional mutation. The computer finds better and better solutions to a problem as species evolve to better adapt to their environments.
Fuzzy systems:	A technique of "continuization," that is, extending concepts to a continuous paradigm, especially for traditionally discrete disciplines such as sets and logic. In ordinary logic, proposition is either true or false, with nothing between, but fuzzy logic allows truthfulness in various degrees.
Rough sets:	A technique of "quantization" and mapping. "Rough" sets means approximation sets. Given a set of elements and attribute values associated with these elements, some of which can be imprecise or incomplete, the theory is suitable to reasoning and discovering relationships in the data.
Chaos:	Nonlinear deterministic dynamical systems that exhibit sustained irregularity and extreme sensitivity to initial conditions.

Background of the five areas

When a computer program solved most of the problems on the final exam for a MIT freshman calculus course in the late 1950s, there was a much excitement for the future of AI. As a result, people thought that one day in the not-too-distant future, the computer might be performing most of the tasks where human intelligence was required. Although this has not occurred, AI has contributed extensively to real world applications. People are, however, still disappointed in the level of achievements of traditional, symbolic AI.

With this background, people have been looking to totally new technologies for some kind of breakthrough. People hoped that neural networks, for example, might provide a breakthrough which was not possible from symbolic AI. There are two major reasons for such a hope. One, neural networks are based upon the brain, and

two, they are based on a totally different philosophy from symbolic AI. Again, no breakthrough that truly simulates human intelligence has occurred. However, neural networks have shown many interesting practical applications that are unique to neural networks, and hence they complement symbolic AI.

Genetic algorithms have a flavor similar to neural networks in terms of dissimilarity from traditional AI. They are computer models based on genetics and evolution. The basic idea is that the genetic program finds better and better solutions to a problem just as species evolve to better adapt to their environments. The basic processes of genetic algorithms are the selection of solutions based on their goodness, the reproduction for crossover of genes, and mutation for random change of genes. Genetic algorithms have been extended in their ways of representing solutions and performing basic processes. A broader definition of genetic algorithms, sometimes called "evolutionary computing," includes not only generic genetic algorithms but also classifier systems, artificial life, and genetic programming where each solution is a computer program. All of these techniques complement symbolic AI.

The story of fuzzy systems is different from those for neural networks and genetic algorithms. Fuzzy set theory was introduced as an extension of ordinary set theory around 1965. But it was known only in a relatively small research community until an industrial application in Japan became a hot topic in 1986. Especially since 1990, massive commercial and industrial applications of fuzzy systems have been developed in Japan, yielding significantly improved performance and cost savings. The situation has been changing as interest in the U.S. rises, and the trend is spreading to Europe and other countries. Fuzzy systems are suitable for uncertain or approximate reasoning, especially for the system with a mathematical model that is difficult to derive.

Rough sets, meaning approximation sets, deviate from the idea of ordinary sets. In fact, both rough sets and fuzzy sets vary from ordinary sets. The area is relatively new and has remained unknown to most of the computing community. The technique is particularly suited to inducing relationships in data. It is compared to other techniques including machine learning in classical AI, Dempster-Shafer theory and statistical analysis, particularly discriminant analysis.

Chaos represents a vast class of dynamical systems that lie between rigid regularity and stochastic randomness. Most scientific and engineering studies and applications have primarily focused on regular phenomena. When systems are not regular, they are often assumed to be random and techniques such as probability theory and statistics are applied. Because of their complexity, chaotic systems have been shunned by most of the scientific community, despite their commonness. Recently, however, there has been growing interest in the practical applications of these systems. Chaos studies those systems that appear random, but the underlying rules are regular.

An additional note: The areas covered in this book are sometimes collectively referred to as *soft computing*. The primary aim of soft computing is close to that of fuzzy systems, that is, to exploit the tolerance for imprecision and uncertainty to achieve tractability, robustness, and low cost in practical applications. I did not use the term soft computing for several reasons. First of all, the term has not been widely recognized and accepted in computer science, even within the AI community. Also it is sometimes confused with "software engineering." And the aim of soft

computing is too narrow for the scopes of most areas. For example, most researchers in neural networks or genetic algorithms would probably not accept that their fields are under the umbrella of soft computing.

Comparisons of the areas covered in this book

For easy understanding of major philosophical differences among the five areas covered in this book, we consider two characteristics: deductive/inductive and numeric/descriptive. With oversimplification, the following table shows typical characteristics of these areas.

	Microscopic, Primarily Numeric	Macroscopic, Descriptive and Numeric
Deductive	Chaos	Fuzzy systems
Inductive	Neural networks Genetic algorithms	Rough sets

In a "deductive" system, rules are provided by experts, and output is determined by applying appropriate rules for each input. In an "inductive" system, rules themselves are induced or discovered by the system rather than by an expert. "Microscopic, primarily numeric" means that the primary input, output, and internal data are numeric. "Macroscopic, descriptive and numeric" means that data involved can be either high level description, such as "very fast," or numeric, such as "100 km/hr."

Both neural networks and genetic algorithms are sometimes referred to as "guided random search" techniques, since both involve random numbers and use some kind of guide such as steepest descent to search solutions in a state space.

Further Reading

For practical applications of AI, both in traditional and newer areas, the following five special issues provide a comprehensive survey.

T. Munakata (Guest Editor), Special Issue on "Commercial and Industrial AI," *Communications of the ACM*, Vol. 37, No. 3, March, 1994.

T. Munakata (Guest Editor), Special Issue on "New Horizons in Commercial and Industrial AI," *Communications of the ACM*, Vol. 38, No. 11, Nov., 1995.

U. M. Fayyad, et al. (Eds.), Data Mining and Knowledge Discovery in Databases, *Communications of the ACM*, Vol. 39, No. 11, Nov., 1996.

T. Munakata (Guest Editor), Special Section on "Knowledge Discovery," *Communications of the ACM*, Vol. 42, No. 11, Nov., 1999.

U. M. Fayyad, et al. (Eds.), Evolving Data Mining into Solutions for Insights, , *Communications of the ACM*, Vol. 45, No. 8, Aug., 2002.

The following four books are primarily for traditional AI, the counterpart of this book.

G. Luger, *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*, 5th Ed., Addison-Wesley; 2005.
S. Russell and P. Norvig, *Artificial Intelligence: Modern Approach*, 2nd Ed., Prentice-Hall, 2003.
E. Rich and K. Knight, *Artificial Intelligence*, 2nd Ed., McGraw-Hill, 1991.
P.H. Winston, *Artificial Intelligence*, 3rd Ed., Addison-Wesley, 1992.

2 Neural Networks: Fundamentals and the Backpropagation Model

2.1 What Is a Neural Network?

A **neural network** (NN) is an abstract computer model of the human brain. The human brain has an estimated 10^{11} tiny units called **neurons**. These neurons are interconnected with an estimated 10^{15} links. Although more research needs to be done, the neural network of the brain is considered to be the fundamental functional source of intelligence, which includes perception, cognition, and learning for humans as well as other living creatures.

Similar to the brain, a neural network is composed of artificial neurons (or units) and interconnections. When we view such a network as a graph, neurons can be represented as nodes (or vertices), and interconnections as edges.

Although the term "neural networks" (NNs) is most commonly used, other names include *artificial neural networks* (*ANNs*)—to distinguish from the natural brain neural networks—*neural nets*, *PDP*(*Parallel Distributed Processing*) *models* (since computations can typically be performed in both parallel and distributed processing), *connectionist models*, and *adaptive systems*.

I will provide additional background on neural networks in a later section of this chapter; for now, we will explore the core of the subject.

2.2 A Neuron

The basic element of the brain is a natural neuron; similarly, the basic element of every neural network is an artificial neuron, or simply **neuron**. That is, a neuron is the basic building block for all types of neural networks.

Description of a neuron

A neuron is an abstract model of a natural neuron, as illustrated in Figs. 2.1. As we can see in these figures, we have inputs x_1, x_2, ..., x_m coming into the neuron. These inputs are the stimulation levels of a natural neuron. Each input x_i is multiplied by its

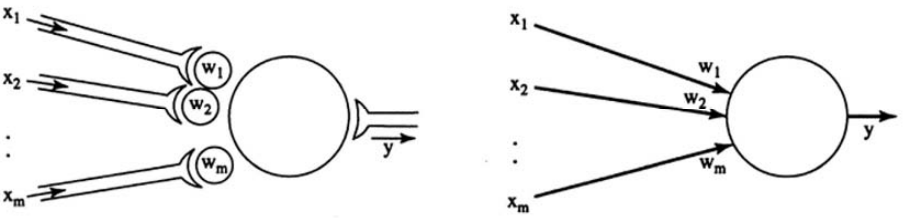


Fig. 2.1 (a)A neuron model that retains the image of a natural neuron. (b) A further abstraction of Fig. (a).

corresponding weight w_i, then the product $x_i w_i$ is fed into the body of the neuron. The weights represent the biological synaptic strengths in a natural neuron. The neuron adds up all the products for $i = 1, m$. The weighted sum of the products is usually denoted as *net* in the neural network literature, so we will use this notation. That is, the neuron evaluates $net = x_1w_1 + x_2w_2 + ... + x_mw_m$. In mathematical terms, given two vectors $\mathbf{x} = (x_1, x_2, ..., x_m)$ and $\mathbf{w} = (w_1, w_2, ..., w_m)$, *net* is the dot (or scalar) product of the two vectors, $\mathbf{x}\cdot\mathbf{w} \equiv x_1w_1 + x_2w_2 + ... + x_mw_m$. Finally, the neuron computes its output y as a certain function of *net*, i.e., $y = f(net)$. This function is called the **activation** (or sometimes **transfer**) **function**. We can think of a neuron as a sort of black box, receiving input vector **x** then producing a scalar output y. The same output value y can be sent out through multiple edges emerging from the neuron.

Activation functions

Various forms of activation functions can be defined depending on the characteristics of applications. The following are some commonly used activation functions (Fig. 2.2).

For the backpropagation model, which will be discussed next, the form of Fig. 2.2 (f) is most commonly used. As a neuron is an abstract model of a brain neuron, these activation functions are abstract models of electrochemical signals received and transmitted by the natural neuron. A threshold shifts a critical point of the *net* value for the excitation of the neuron.

2.3 Basic Idea of the Backpropagation Model

Although many neural network models have been proposed, the backpropagation is the most widely used model in terms of practical applications. No statistical surveys have been conducted, but probably over 90% of commercial and industrial applications of neural networks use backpropagation or its derivatives. We will study the fundamentals of this popular model in two major steps. In this section, we will present a basic outline. In the next Section 2.4, we will discuss technical details. In Section 2.5, we will describe a so-called cookbook recipe summarizing the resulting

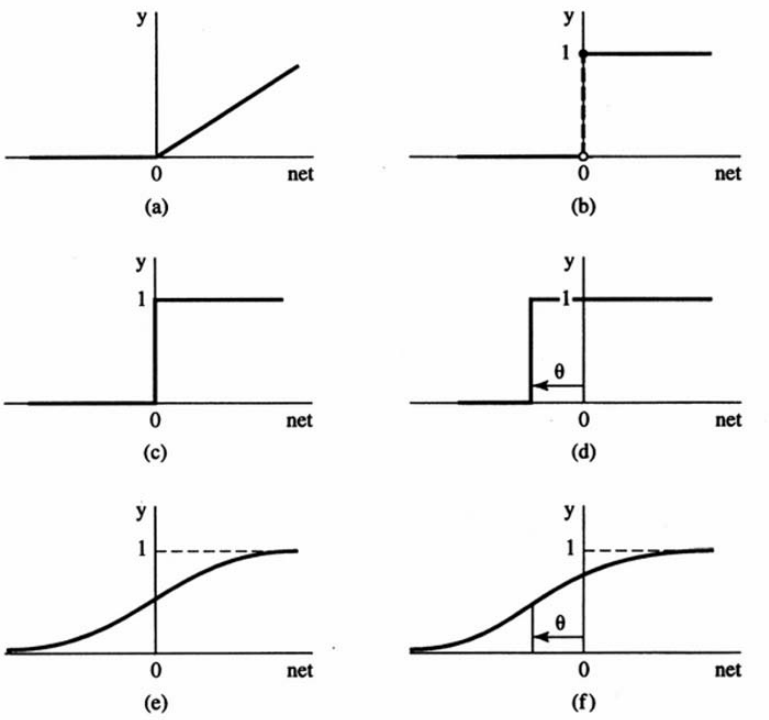


Fig. 2.2 (a) A piecewise linear function: $y = 0$ for $net < 0$ and $y = k \cdot net$ for $net \geq 0$, where k is a positive constant. (b) A step function: $y = 0$ for $net < 0$ and $y = 1$ for $net \geq 0$. (c) A conventional approximation graph for the step function defined in (b). This type of approximation is common practice in the neural network literature. More precisely, this graph can be represented by one with a steep line around $net = 0$, e.g., $y = 0$ for $net < -\varepsilon$, $y = (net - \varepsilon)/2\varepsilon + 1$ for $-\varepsilon \leq net < \varepsilon$, and $y = 1$ for $net \geq \varepsilon$, where ε is a very small positive constant, that is, $\varepsilon \rightarrow +0$. (d) A step function with threshold θ: $y = 0$ for $net + \theta < 0$ and $y = 1$ otherwise. The same conventional approximation graph is used as in (c). Note that in general, a graph where net is replaced with $net + \theta$ can be obtained by shifting the original graph without threshold horizontally by θ to the left. (This means that if θ is negative, shift by $|\theta|$ to the right.) Note that we can also modify Fig. 2.2 (a) with threshold. (e) **A sigmoid function**: $y = 1/[1 + \exp(-net)]$, where $\exp(x)$ means e^x. (f) **A sigmoid function with threshold** θ: $y = 1/[1 + \exp\{-(net + \theta)\}]$.

formula necessary to implement neural networks.

Architecture

The pattern of connections between the neurons is generally called the **architecture** of the neural network. The backpropagation model is one of **layered neural networks**, since each neural network consists of distinct layers of neurons. Fig. 2.3 shows a simple example. In this example, there are three layers, called **input**, **hidden**,

and **output layers**. In this specific example, the input layer has four neurons, hidden has two, and output has three.

Generally, there are one input, one output, and any number of hidden layers. One hidden layer as in Fig. 2.3 is most common; the next common numbers are zero (i.e., no hidden layer) and two. Three or more hidden layers are very rare. You may remember that to count the total number of layers in a neural network, some authors include the input layer while some don't. In the above example, the numbers will be 3 and 2, respectively, in these two ways of counting. The reason the input layer is sometimes not counted is that the "neurons" in the input layer do not compute anything. Their function is merely to send out input signals to the hidden layer neurons. A less ambiguous way of counting the number of layers would be to count the number of hidden layers. Fig. 2.3 is an example of a neural network with one hidden layer.

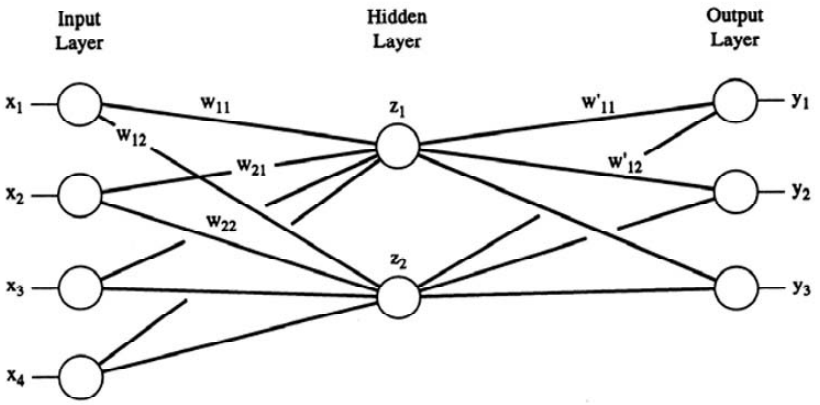


Fig. 2.3 simple example of backpropagation architecture. Only selected weights are illustrated.

The number of neurons in the above example, 4, 2, and 3, is much smaller than the ones typically found in practical applications. The number of neurons in the input and output layers are usually determined from a specific application problem. For example, for a written character recognition problem, each character is plotted on a two-dimensional grid of 100 points. The number of input neurons would then be 100. For the hidden layer(s), there are no definite numbers to be computed from a problem. Often, the trial-and-error method is used to find a good number.

Let us assume one hidden layer. All the neurons in the input layer are connected to all the neurons in the hidden layer through the edges. Similarly, all the neurons in the hidden layer are connected to all the neurons in the output layer through the edges. Suppose that there are n_i, n_h, and n_o neurons in the input, hidden, and output layers, respectively. Then there are $n_i \times n_h$ edges from the input to hidden layers, and $n_h \times n_o$ edges from the hidden to output layers.

A weight is associated with each edge. More specifically, weight w_{ij} is associated with the edge from input layer neuron x_i to hidden layer neuron z_j; weight w'_{ij} is

associated with the edge from hidden layer neuron z_i to output layer neuron y_j. (Some authors denote w_{ij} as w_{ji} and w'_{ij} as w'_{ji}, i.e., the order of the subscripts are reversed. We follow graph theory convention that a directed edge from node i to node j is represented by e_{ij}.) Typically, these *weights are initialized randomly* within a specific range, depending on the particular application. For example, weights for a specific application may be initialized randomly between -0.5 and +0.5. Perhaps $w_{11} = 0.32$ and $w_{12} = -0.18$.

The input values in the input layer are denoted as $x_1, x_2, ..., x_{ni}$. The neurons themselves can be denoted as 1, 2, ... n_i, or sometimes $x_1, x_2, ..., x_{ni}$, the same notation as input. (Different notations can be used for neurons as, for example, $u_{x1}, u_{x2}, ..., u_{xni}$, but this increases the number of notations. We would like to keep the number of notations down as long as they are practical.) These values can collectively be represented by the input vector $\mathbf{x} = (x_1, x_2, ..., x_{ni})$. Similarly, the neurons and the internal output values from these neurons in the hidden layer are denoted as $z_1, z_2, ..., z_{nh}$ and $\mathbf{z} = (z_1, z_2, ..., z_{nh})$. Also, the neurons and the output values from the neurons in the output layer are denoted as $y_1, y_2, ..., y_{no}$ and $\mathbf{y} = (y_1, y_2, ..., y_{no})$. Similarly, we can define weight vectors; e.g., $\mathbf{w}_j = (w_{1j}, w_{2j}, ..., w_{ni,j})$ represents the weights from all the input layer neurons to the hidden layer neuron z_j; $\mathbf{w'}_j = (w'_{1j}, w'_{2j}, ..., w'_{nh,j})$ represents the weights from all the hidden layer neurons to the output layer neuron y_j. We can also define the weight matrices $\mathbf{W}$ and $\mathbf{W'}$ to represent all the weights in a compact way as follows:

$$\mathbf{W} = [\mathbf{w}_1^T \mathbf{w}_2^T ... \mathbf{w}_{nh}^T] = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1,nh} \\ w_{21} & w_{22} & \dots & w_{2,nh} \\ & \dots & & \\ w_{ni,1} & & \dots & w_{ni,nh} \end{bmatrix}$$

where $\mathbf{w}^T$ means the transpose of $\mathbf{w}$, i.e., when $\mathbf{w}$ is a row vector, $\mathbf{w}^T$ is the column vector of the same elements. Matrix $\mathbf{W'}$ can be defined in the same way for vectors $\mathbf{w'}_j$.

When there are two hidden layers, the above can be extended to $\mathbf{z} = (z_1, z_2, ..., z_{nh})$ and $\mathbf{z'} = (z'_1, z'_2, ..., z'_{nh'})$, where $\mathbf{z}$ represents the first hidden layer and $\mathbf{z'}$ the second. When there are three or more hidden layers, these can be extended to $\mathbf{z}$, $\mathbf{z'}$, $\mathbf{z''}$, and so on. But since three or more hidden layers are very rare, we normally do not have to deal with such extensions. The weights can also be extended similarly. When there are two hidden layers, the weight matrices, for example, can be extended to: $\mathbf{W}$ from the input to first hidden layers, $\mathbf{W'}$ from the first to second hidden layers, and $\mathbf{W''}$ from the second hidden to output layers.

Learning (training) process

Having set up the architecture, the neural network is ready to *learn*, or said another way, we are ready to *train* the neural network. A rough sketch of the learning process is presented in this section. More details will be provided in the next section.

A neural network learns patterns by adjusting its weights. Note that "patterns" here should be interpreted in a very broad sense. They can be visual patterns such as

two-dimensional characters and pictures, as well as other patterns which may represent information in physical, chemical, biological, or management problems. For example, acoustic patterns may be obtained by taking snapshots at different times. Each snapshot is a pattern of acoustic input at a specific time; the abscissa may represent the frequency of sound, and the ordinate, the intensity of the sound. A pattern in this example is a graph of an acoustic spectrum. To predict the performance of a particular stock in the stock market, the abscissa may represent various parameters of the stock (such as the price of the stock the day before, and so on), and the ordinate, values of these parameters.

A neural network is given correct pairs of (input pattern, target output pattern). Hereafter we will call the target output pattern simply **target pattern**. That is, (input pattern 1, target pattern 1), (input pattern 2, target pattern 2), and so forth, are given. Each target pattern can be represented by a target vector $\mathbf{t} = (t_1, t_2, ..., t_{no})$. The learning task of the neural network is to adjust the weights so that it can output the target pattern for each input pattern. That is, when input pattern 1 is given as input vector **x**, its output vector **y** is equal (or close enough) to the target vector **t** for target pattern 1; when input pattern 2 is given as input vector **x**, its output vector **y** is equal (or close enough) to the target vector **t** for target pattern 2; and so forth.

When we view the neural network macroscopically as a black box, it learns *mapping* from the input vectors to the target vectors. Microscopically it learns by adjusting its weights. As we see, in the backpropagation model we assume that there is a *teacher* who knows and tells the neural network what are correct input-to-output mapping. The backpropagation model is called a **supervised learning** method for this reason, i.e., it learns under supervision. It cannot learn without being given correct sample patterns.

The learning procedure can be outlined as follows:

Outline of the learning (training) algorithm

Outer loop. Repeat the following until the neural network can consecutively map all patterns correctly.

Inner loop. For each pattern, repeat the following Steps 1 to 3 until the output vector **y** is equal (or close enough) to the target vector **t** for the given input vector **x**.

Step 1. Input **x** to the neural network.

Step 2. *Feedforward*. Go through the neural network, from the input to hidden layers, then from the hidden to output layers, and get output vector **y**.

Step 3. *Backward propagation of error corrections*. Compare **y** with **t**. If **y** is equal or close enough to **t**, then go back to the beginning of the Outer loop. Otherwise, backpropagate through the neural network and adjust the weights so that the next **y** is closer to **t**, then go back to the beginning of the Inner loop.

In the above, each *Outer loop* iteration is called an **epoch**. An epoch is one cycle through the entire set of patterns under consideration. Note that to terminate the

outer loop (i.e., the entire algorithm), the neural network must be able to produce the target vector for any input vector. Suppose, for example, that we have two sample patterns to train the neural network. We repeat the inner loop for Sample 1, and the neural network is then able to map the correct **t** after, say, 10 iterations. We then repeat the inner loop for Sample 2, and the neural network is then able to map the correct **t** after, say, 8 iterations. This is the end of the first epoch. The end of the first epoch is not usually the end of the algorithm or outer loop. After the training session for Sample 2, the neural network "forgets" part of what it learned for Sample 1. Therefore, the neural network has to be trained again for Sample 1. But, the second round (epoch) training for Sample 1 should be shorter than the first round, since the neural network has not completely forgotten Sample 1. It may take only 4 iterations for the second epoch. We can then go to Sample 2 of the second epoch, which may take 3 iterations, and so forth. When the neural network gives correct outputs for both patterns with 0 iterations, we are done. This is why we say "consecutively map all patterns" in the first part of the algorithm. Typically, many epochs are required to train a neural network for a set of patterns.

There are alternate ways of performing iterations. One variation is to train Pattern 1 until it converges, then store its w_{ij}s in temporary storage without actually updating the weights. Repeat this process for Patterns 2, 3, and so on, for either several or the entire set of patterns. Then take the average of these weights for different patterns for updating. Another variation is that instead of performing the inner loop iterations until one pattern is learned, the patterns are given in a row, one iteration for each pattern. For example, one iteration of Steps 1, 2, and 3 are performed for Sample 1, then the next iteration is immediately performed for Sample 2, and so on. Again, all samples must converge to terminate the entire iteration.

Case study - pattern recognition of hand-written characters

For easy understanding, let us consider a simple example where our neural network learns to recognize hand-written characters. The following Fig. 2.4 shows two sample input patterns ((a) and (b)), a target pattern ((c)), input vector $\boldsymbol{x}$ for pattern (a) ((d)), and layout of input, output, and target vectors ((e)). When people hand-write characters, often the characters are off from the standard ideal pattern. The objective is to make the neural network learn and recognize these characters even if they are slightly deviated from the ideal pattern.

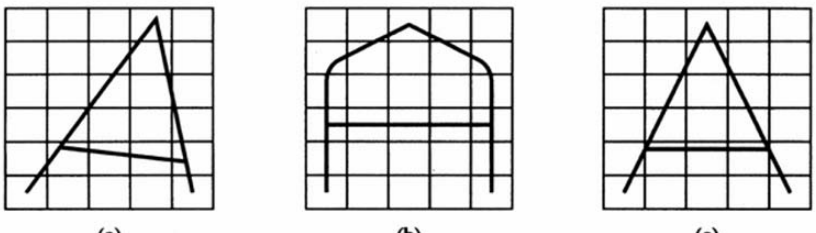
Each pattern in this example is represented by a two-dimensional grid of 6 rows and 5 columns. We convert this two-dimensional representation to one-dimensional by assigning the top row squares to x_1 to x_5, the second row squares to x_6 to x_{10}, etc., as shown in Fig. (e). In this way, two-dimensional patterns can be represented by the one-dimensional layers of the neural network. Since x_i ranges from $i = 1$ to 30, we have 30 input layer neurons. Similarly, since y_i also ranges from $i = 1$ to 30, we have 30 output layer neurons. In this example, the number of neurons in the input and output layers is the same, but generally their numbers can be different. We may arbitrarily choose the number of hidden layer neurons as 15.

The input values of x_i are determined as follows. If a part of the pattern is within the square x_i, then $x_i = 1$, otherwise $x_i = 0$. For example, for Fig. (c), $x_1 = 0$, $x_2 = 0$, $x_3 = 1$, etc. Fig. 2.4 representation is coarse since this example is made very simple for illustrative purpose. To get a finer resolution, we can increase the size of the grid to,

e.g., 50 rows and 40 columns.

After designing the architecture, we initialize all the weights associated with edges randomly, say, between -0.5 and 0.5. Then we perform the training algorithm described before until both patterns are correctly recognized. In this example, each y_i may have a value between 0 and 1. 0 means a complete blank square, 1 means a complete black square, and a value between 0 and 1 means a "between" value: gray. Normally we set up a threshold value, and a value within this threshold value is considered to be close enough. For example, a value of y_i anywhere between 0.95 and 1.0 may be considered to be close enough to 1; a value of y_i anywhere between 0.0 and 0.05 may be considered to be close enough to 0.

After completing the training sessions for the two sample patterns, we might have a surprise. The trained neural network gives correct answers not only for the sample data, but also it may give correct answers for totally new similar patterns. In other words, the neural network has robustness for identifying data. This is indeed a major goal of the training - a neural network can generalize the characteristics associated with the training examples and recognize similar patterns it has never been given before.



0	0	0	1	0
0	0	1	1	0
0	0	1	1	1
0	1	0	0	1
1	1	1	1	1
1	0	0	0	1

(d)

x_1	x_2	x_3	x_4	x_5
x_6	x_7	x_8	x_9	x_{10}
x_{11}	x_{12}	x_{13}	x_{14}	x_{15}
x_{16}	x_{17}	x_{18}	x_{19}	x_{20}
x_{21}	x_{22}	x_{23}	x_{24}	x_{25}
x_{26}	x_{27}	x_{28}	x_{29}	x_{30}

(e)

Fig. 2.4(a) and (b): two sample input patterns; (c): a target pattern; (d) input vector **x** for Pattern (a); (e) layout for input vector **x**, output vector **y** (for **y**, replace *x* with *y*), and target vector **t** (for **t**, replace *x* with *t*);

We can further extend this example to include more samples of character "A," as well as to include additional characters such as "B," "C," and so on. We will have training samples and ideal patterns for these characters. However, a word of caution for such extensions in general: training of a neural network for many patterns is not a trivial matter, and it may take a long time before completion. Even worse, it may

never converge to completion. It is not uncommon that training a neural network for a practical application requires hours, days, or even weeks of continuous running of a computer. Once it is successful, even if it takes a month of continuous training, it can be copied to other systems easily and the benefit can be significant.

2.4 Details of the Backpropagation Model

Having understood the basic idea of the backpropagation model, we now discuss technical details of the model. With this material, we will be able to design neural networks for various application problems and write computer programs to obtain solutions.

In this section, we describe how such a formula can be derived. In the next section, we will describe a so-called cookbook recipe summarizing the resulting formula necessary to implement neural networks. If you are in a hurry to implement a neural network, or cannot follow some of the mathematical derivations, the details of this section can be skipped. However, it is advisable to follow the details of such basic material once for two reasons. One, you will get a much deeper understanding of the material. Two, if you have any questions on the material or doubts about typos in the formula, you can always check them yourself.

Architecture

The network architecture is a generalization of a specific example discussed before in Fig. 2.3, as shown in Fig. 2.5. As before, this network has three layers: input, hidden, and output. Networks with these three layers are the most common. Other forms of network configurations such as no or two hidden layers can be handled similarly.

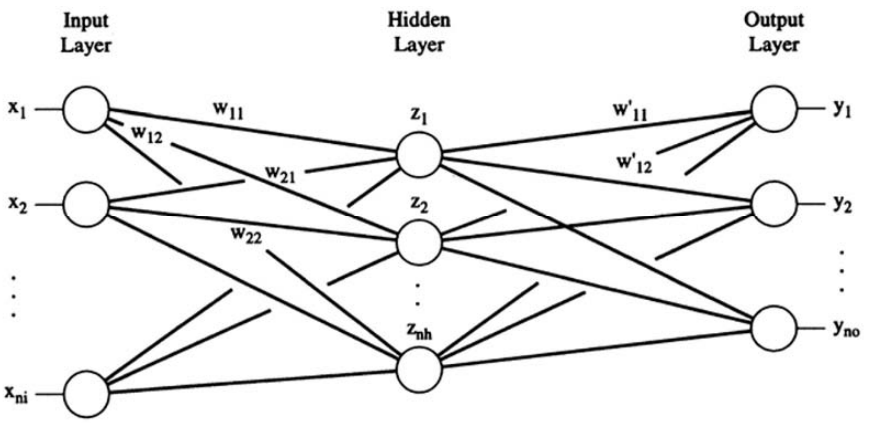


Fig. 2.5 A general configuration of the backpropagation model neural network.

There are n_i, n_h, and n_o neurons in the input, hidden, and output layers, respectively. Weight w_{ij} is associated to the edge from input layer neuron x_i to hidden layer neuron z_j; weight w'_{ij} is associated to the edge from hidden layer neuron z_i to output layer neuron y_j. As discussed before, the neurons in the input layer as well as input values at these neurons are denoted as $x_1, x_2, ..., x_{ni}$. These values can collectively be represented by the input vector $\mathbf{x} = (x_1, x_2, ..., x_{ni})$. Similarly, the neurons and the internal output values from neurons in the hidden layer are denoted as $z_1, z_2, ..., z_{nh}$, and $\mathbf{z} = (z_1, z_2, ..., z_{nh})$. Also, the neurons and the output values from the neurons in the output layer are denoted as $y_1, y_2, ..., y_{no}$, and $\mathbf{y} = (y_1, y_2, ..., y_{no})$. Similarly, we can define weight vectors; e.g., $\mathbf{w}_j = (w_{1j}, w_{2j}, ..., w_{ni,j})$ represents the weights from all the input layer neurons to the hidden layer neuron z_j; $\mathbf{w'}_j = (w'_{1j}, w'_{2j}, ..., w'_{nh,j})$ represents the weights from all the hidden layer neurons to the output layer neuron y_j.

Initialization of weights

Typically these weights are initialized randomly within a certain range, a specific range depending on a particular application. For example, weights for a specific application may be initialized with uniform random numbers between -0.5 and +0.5.

Feedforward: Activation function and computing z from x, and y from z

Let us consider a local neuron j, which can represent either a hidden layer neuron z_j or an output layer neuron y_j. (As an extension, if there is a second hidden layer, j can also represent a second hidden layer neuron z'_j.) (Fig. 2.6).

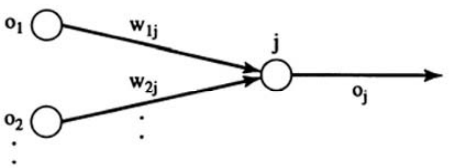


Fig. 2.6. Configuration of a local neuron j.

The weighted sum of incoming activations or inputs to neuron j is: $net_j = \Sigma_i w_{ij} o_i$, where Σ_i is taken over all i values of the incoming edges. (Here o_i is the *input* to neuron j. Although i_i may seem to be a better notation, o_i is originally the output of neuron i. Rather than using two notations for the same thing and equating o_i to i_i in every computation, we use a single notation o_i.) Output from neuron j, o_j, is an activation function of net_j. Here we use the most commonly used form of activation function, sigmoid with threshold (Fig. 2.2 (f)), as $o_j = f_j(net_j) = 1/[1 + \exp\{-(net_j + \theta_j)\}]$. (Determining the value of θ_j will be discussed shortly.)

Given input vector $\mathbf{x}$, we can compute vectors $\mathbf{z}$ and $\mathbf{y}$ using the above formula. For example, to determine z_j, compute $net_j = \Sigma_i w_{ij} x_i$, then $z_j = f_j(net_j) = 1/[1 + \exp\{-(net_j + \theta_j)\}]$. In turn, these computed values of z_j are used as incoming activations or inputs to neurons in the output layer. To determine y_j, compute $net_j =$

$\Sigma_i\, w'_{ij}z_i$, then $y_j = f_j(net_j) = 1/[1 + \exp\{-(net_j + \theta'_j)\}]$. (Determining the value of θ'_j will also be discussed shortly.) We note that for example, vector **net** = (net_1, net_2 . . .) = ($\Sigma_i\, w_{i1}x_i$, $\Sigma_i\, w_{i2}x_i$, . . .) = ($\Sigma_i\, x_i w_{i1}$, $\Sigma_i\, x_i w_{i2}$, . . .) can be represented in compact way as **net** = **xW**, where **xW** is the matrix product of **x** and **W**.

Backpropagation for adjusting weights to minimize the difference between output y and target t

We now adjust the weights in such a way as to make output vector **y** closer to target vector **t**. We perform this starting from the output layer back to the hidden layer, modifying the weights, **W'**, then further backing from the hidden layer to the input layer, and changing the weights, **W**. Because of this backward changing process of the weights, the model is named "**backpropagation**." This scheme is also called the **generalized delta rule** since it is a generalization of another historically older procedure called the delta rule. Note that the backward propagation is only for the purpose of modifying the weights. In the backpropagation model discussed here, activation or input information to neurons advances only forward from the input to hidden, then from the hidden to output layers. The activation information never goes backward, for example from the output to hidden layers. There are neural network models in which such backward information passing occurs as feedback. This category of models is called **recurrent** neural networks. The meaning of the backward propagation of the backpropagation model should not be confused with these recurrent models.

To consider the difference of the two vectors **y** and **t**, we take the *square* of the error or "distance" of the two vectors as follows.

$$E = \left(\frac{1}{2}\right)\sum_j \left(t_j - y_j\right)^2$$

Generally, taking the square of the difference is a common approach for many minimization problems. Without taking the square, e.g., $E = \Sigma_j\, (t_j - y_j)$, positive and negative values of $(t_j - y_j)$ for different j's cancel out and E will become smaller than the actual error. Summing up the absolute differences, i.e., $E = \Sigma_j\, |t_j - y_j|$ is correct, but taking the square is usually easier for computation. The factor (1/2) is also a common practice when the function to be differentiated has the power of 2; after differentiation, the original factor of (1/2) and the new factor of 2 from differentiation cancel out, and the coefficient of the derivative will become 1.

The goal of the learning procedure is to minimize E. We want to the reduce the error E by improving the current values of w_{ij} and w'_{ij}. In the following derivation of the backpropagation formula, we assume that w_{ij} can be either w_{ij} and w'_{ij}, unless otherwise specified. Improving the current values of w_{ij} is performed by adding a small fraction of w_{ij}, denoted as Δw_{ij}, to w_{ij}. In equation form:

$$w_{ij}^{(n+1)} = w_{ij}^{(n)} + \Delta w_{ij}^{(n)}.$$

Here the superscript (n) in $w_{ij}^{(n)}$ represents the value of w_{ij} at the n-th iteration. That is, the equation says that the value of w_{ij} at the $(n+1)$st iteration is obtained by adding

the values of w_{ij} and Δw_{ij} at the n-th iteration.

Our next problem is to determine the value of $\Delta w_{ij}^{(n)}$. This is done by using the steepest descent of E in terms of w_{ij}, i.e., $\Delta w_{ij}^{(n)}$ is set proportional to the gradient of E. (This is a very common technique used for minimization problems, called the **steepest descent method**.) As an analogy, we can think of rolling down a small ball on a slanted surface in a three-dimensional space. The ball will fall in the steepest direction to reduce the gravitational potential most, analogous to reducing the error E most. To find the steepest direction of the gravitational potential, we compute $-(\partial E/\partial x)$ and $-(\partial E/\partial y)$, where E is the gravitational potential determined by the surface; $-(\partial E/\partial x)$ gives the gradient in the x direction and $-(\partial E/\partial y)$ gives the gradient in the y direction.

From calculus, we remember that the symbol "∂" denotes partial differentiation. To partially differentiate a function of multiple variables with respect to a specific variable, we consider the other remaining variables as if they were constants. For example, for $f(x, y) = x^2y^5 + e^{-x}$, we have $\partial f/\partial y = 5x^2y^4$, the term e^{-x} becoming zero for partial differentiation with respect to y. In our case, the error E is a function of w_{ij} and w'_{ij} rather than only two variables, x and y, in a three-dimensional space. The number of w_{ij}'s is equal to the number of edges from the input to hidden layers, and the number of w'_{ij}'s is equal to the number of edges from the hidden to output layers. To make $\Delta w_{ij}^{(n)}$ proportional to the gradient of E, we set $\Delta w_{ij}^{(n)}$ as:

$$\Delta w_{ij}^{(n)} = -\eta \left(\frac{\partial E}{\partial w_{ij}} \right)$$

where η is a positive constant called the **learning rate**.

Our next step is to compute $\partial E/\partial w_{ij}$. Fig. 2.7 shows the configuration under consideration. We are to modify the weight w_{ij} (or w'_{ij}) associated to the edge from neuron i to neuron j. The "output" (activation level) from neuron i (i.e., the "input" to neuron j) is o_i, and the "output" from neuron j is o_j. Note again that symbol "o" is used for both output and input, since output from one neuron becomes input to the neurons in the downstream. For a neural network with one hidden layer, when i is an input layer neuron, o_i is x_i and o_j is z_j; when i is a hidden layer neuron, o_i is z_i and o_j is y_j.

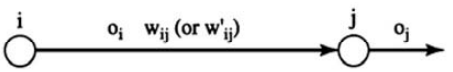


Fig. 2.7. The associated weight and "output" to neurons i and j.

Note that E is a function of y_j as $E = (1/2)\ \Sigma_j\ (t_j - y_j)^2$; in turn, y_j is a function (an activation function) of net_j as $y_j = f(net_j)$, and again in turn net_j is a function of w_{ij} as $net_j = \Sigma_i\ w_{ij}o_i$, where o_i is the incoming activation from neuron i to neuron j. By using the calculus chain rule we write:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial net_j}\frac{\partial net_j}{\partial w_{ij}}.$$

The second factor $\partial net_j/\partial w_{ij}$ requires only a few steps to be evaluated:

$$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial\left(\sum_k w_{kj}o_k\right)}{\partial w_{ij}} \qquad \text{(by substituting } net_j = \Sigma_k\, w_{kj}o_k\text{)}$$

$$= \frac{\partial w_{1j}o_1}{\partial w_{ij}} + \frac{\partial w_{2j}o_2}{\partial w_{ij}} + \ldots + \frac{\partial w_{ij}o_i}{\partial w_{ij}} + \ldots$$

$= o_i$. (partial differentiation; all terms are 0 except the term for $k = i$. This type of partial differentiation appears in many applications.)

For convenience of computation, we define:

$$\delta_j = -\frac{\partial E}{\partial net_j}$$

The introduction of this new variable, δ_j, turns out to be quite useful for adjusting the weights (which is the key ingredient of the backpropagation model) as we will see in the following. Summarizing the results so far, we have

$$\Delta w_{ij}^{(n)} = \eta\ \delta_j\, o_i,$$

where $\delta_j = -\partial E/\partial net_j$ is yet to be determined. To compute $\partial E/\partial net_j$, we again use the chain rule:

$$\frac{\partial E}{\partial net_j} = \frac{\partial E}{\partial o_j}\frac{\partial o_j}{\partial net_j}$$

An interpretation of the right-hand side expression is that the first factor ($\partial E/\partial o_j$) reflects the change in error E as a function of changes in output o_j of neuron j. The second factor ($\partial o_j/\partial net_j$) represents the change in output o_j as a function of changes in input net_j of neuron j. To compute the first factor $\partial E/\partial o_j$, we consider two cases: (1) neuron j is in the output layer, and (2) j is not in the output layer, i.e., it is in a hidden layer in a multi-hidden layer network, or j is in the hidden layer in a one-hidden layer network.

Case 1. j is in the output layer.

$$\frac{\partial E}{\partial o_j} = \frac{\partial\left\{\left(\frac{1}{2}\right)\sum_k (t_k - o_k)^2\right\}}{\partial o_j} = -(t_j - o_j).$$

Case 2. j is in a hidden layer.

$$\frac{\partial E}{\partial o_j} = \sum_k \frac{\partial E}{\partial net_k}\frac{\partial net_k}{\partial o_j} = \sum_k \frac{\partial E}{\partial net_k}\frac{\partial \sum_\ell w'_{\ell k} o_\ell}{\partial o_j} = \sum_k \frac{\partial E}{\partial net_k} w'_{jk}$$

$$= -\sum_k \delta_k w'_{jk} .$$

The summation Σ_k is taken over the neurons in the *immediately* succeeding layer of neuron j, but not in the layers beyond. This means that if there are two hidden layers in the network, and j is in the first hidden layer, k will be over the neurons in the second hidden layer, but not over the output layer. The partial differentiation with respect to o_j yields only the terms that are directly related to o_j. If there is only one hidden layer, as in our primary analysis, there is no need for such justifications since there is only one succeeding layer which is the output layer.

The second factor, $(\partial o_j/\partial net_j) = (\partial f_j(net_j)/\partial net_j)$, depends on a specific activation function form. When the activation function is the sigmoid with threshold as $o_j = f_j(net_j) = 1/[1 + \exp\{-(net_j + \theta_j)\}]$, we can show: $(\partial f_j(net_j)/\partial net_j) = o_j(1 - o_j)$ as follows. Let $f(t) = 1/[1 + \exp\{-(t + \theta)\}]$. Then by elementary calculus, we have $f'(t) = \exp\{-(t + \theta)\}/[1 + \exp\{-(t + \theta)\}]^2 = \{1/f(t) - 1\} \cdot \{f(t)\}^2 = f(t)\{1 - f(t)\}$. By equating $f(t) = o_j$, we have $o_j(1 - o_j)$ for the answer.

By substituting these results back into $\delta_j = -\partial E/\partial net_j$, we have:

Case 1. j is in the output layer; this case will be used to modify w'_{ij} in our analysis.

$$\delta_j = (t_j - o_j)\frac{\partial f_j(net_j)}{\partial net_j} = (t_j - o_j)o_j(1 - o_j)$$

Case 2. j is in a hidden layer; this case will be used to modify w_{ij} (rather than w'_{ij}) in our analysis.

$$\delta_j = \frac{\partial f_j(net_j)}{\partial net_j} \sum_k \delta_k w'_{jk} = o_j(1 - o_j) \sum_k \delta_k w'_{jk}$$

More on backpropagation for adjusting weights

The momentum term

Furthermore, to stabilize the iteration process, a second term, called the **momentum term**, can be added as follows:

$$\Delta w_{ij}^{(n)} = \eta\ \delta_j\ o_i\ +\ \alpha\ \Delta w_{ij}^{(n-1)}$$

where α is a positive constant called the **momentum rate**. There is no general formula to compute these constants η and α. Often these constant values are determined experimentally, starting from certain values. Common values are, for example, $\eta = 0.5$ and $\alpha = 0.9$.

Adjustment of the threshold θ_j just like a weight

So far we have not discussed how to adjust the threshold θ_j in the function $f_j(net_j) = 1/[1 + \exp\{-(net_j + \theta_j)\}]$. This can be done easily by using a small trick; the thresholds can be learned just like any other weights. Since $net_j + \theta_j = \Sigma_i w_{ij}o_i + (\theta_j \times 1)$, we can treat θ_j as if the weight associated to an edge from an imaginary neuron to neuron j, where the output (activation) o_i of the imaginary neuron is always 1. Fig. 2.8 shows this technique. We can also denote θ_j as $w_{m+1,j}$.

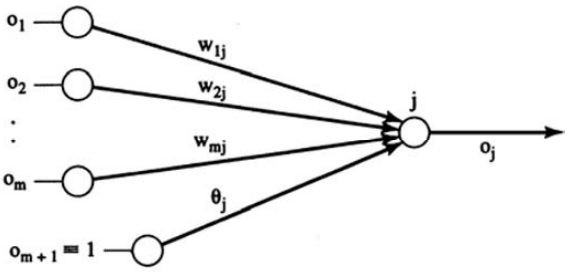


Fig. 2.8. Treating θ_j as if the weight associated to an imaginary edge.

Multiple patterns

So far we have assumed that there is only one pattern. We can extend our preceding discussions to multiple patterns. We can add subscript p, which represents a specific pattern p, to the variables defined previously. For example, E_p represents the error for pattern p. Then

$$E = \Sigma_p E_p$$

gives the total error for all the patterns. Our problem now is to minimize this total error, which is the sum of errors for individual patterns. This means the gradients are averaged over all the patterns. In practice, it is usually sufficient to adjust the weights considering one pattern at a time, or averaging over several patterns at a time.

Iteration termination criteria

There are two commonly used criteria to terminate iterations. One is to check $|t_j - y_j| \leq \varepsilon$ for every j, where ε is a certain preset small positive constant. Another criterion is $E = (1/2)\,\Sigma_j\,(t_j - y_j)^2 \leq \varepsilon$, where again ε is a preset small positive constant. A value of ε for the second criterion would be larger than a value of ε for the first one, since the second criterion checks the total error. The former is more accurate, but often the second is sufficient for practical applications. In effect, the second criterion checks the average error for the whole pattern.

2.5 A Cookbook Recipe to Implement the Backpropagation Model

We assume that our activation function is the sigmoid with threshold θ. We also assume that network configuration has one hidden layer, as shown in Fig. 2.9. Note that there is an additional (imaginary) neuron in each of the input and hid den layers. The threshold θ is treated as if it were the weight associated to the edge from the imaginary neuron to another neuron. Hereafter, "weights" include these thresholds θ. We can denote θ_j as $w_{ni+1,j}$ and θ'_j as $w_{nh+1,j}$.

Initialization.

Initialize all the weights, including the thresholds, to small uniform random numbers, e.g., between -0.5 and +0.5.

Iterations.

Perform iterations until all patterns are learned (good luck!). There are different ways of performing iterations; one simple way is to perform inner-loop iterations for each pattern at a time (as discussed in Section 2.3).

Outer loop. Repeat the following until the neural network can consecutively map all patterns correctly.

Inner loop. For each pattern, repeat the following Steps 1 to 3 until the output vector **y** is equal or close enough to the target vector **t** for the given input vector **x**. Use an iteration termination criterion.

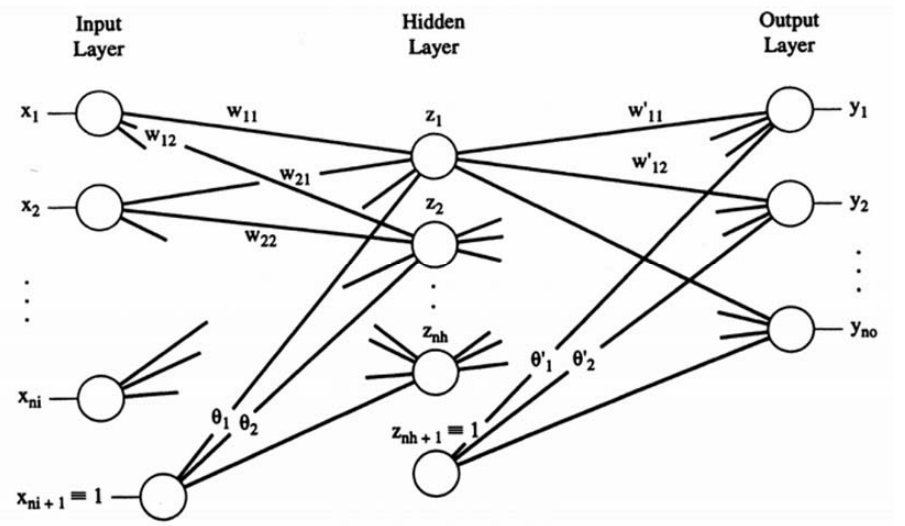


Fig. 2.9. A network configuration for the sigmoid activation function with threshold.

Step 1. Input **x** to the neural network.

Step 2. *Feedforward.* Go through the neural network, from the input to hidden layers, then from the hidden to output layers, and get output vector **y**.

Step 3. *Backward propagation for error corrections.* Compare **y** with **t**. If **y** is equal or close enough to **t**, then go back to the beginning of the Outer loop. Otherwise, backpropagate through the neural network and adjust the weights so that the next **y** is closer to **t** (see the next backpropagation process), then go back to the beginning of the Inner loop.

Backpropagation Process in Step 3.

Modify the values of w_{ij} and w'_{ij} according to the following recipe.

$$w_{ij}^{(n+1)} = w_{ij}^{(n)} + \Delta w_{ij}^{(n)}.$$

where,

$$\Delta w_{ij}^{(n)} = \eta\ \delta_j\ o_i\ +\ \alpha\ \Delta w_{ij}^{(n-1)}$$

where η and α are positive constants. (For the first iteration, assume the second term is zero.) Since there is no general formula to compute these constants η and α, start with arbitrary values, for example, $\eta = 0.5$ and $\alpha = 0.9$. We can adjust the values of η and α during our experiment for better convergence.

Assuming that the activation function is the sigmoid with threshold θ_j, then δ_j can be determined as follows:

Case 1. j is in the output layer; modify w'_{ij}.

$$\delta_j = (t_j - o_j)o_j(1 - o_j)$$

Case 2. j is in the hidden layer; modify w_{ij}.

$$\delta_j = o_j(1 - o_j)\ \Sigma_k\ \delta_k\ w'_{jk}$$

Note that δ_k in Case 2 has been computed in Case 1. The summation Σ_k is taken over $k = 1$ to n_o on the output layer.

Programming considerations

Writing a short program and running it for a simple problem such as Fig. 2.4 is a good way to understand and test the backpropagation model. Although Fig. 2.4 deals with character recognition, the basics of the backpropagation model is the same for other problems. That is, we can apply the same technique for many types of applications.

The figures in Fig. 2.4 are given for the purpose of illustration, and they are too coarse to adequately represent fine structures of the characters. We may increase the

mesh size from 6×5 to 10×8 or even higher depending on the required resolution. We may train our neural network for two types of characters, e.g., "A"and "M". We will provide a couple of sample patterns and a target pattern for each character. In addition, we may have test patterns for each character. These test patterns are not used for training, but will be given to test whether the neural network can identify these characters correctly.

A conventional high level language, such as *C* or Pascal, can be used for coding. Typically, this program requires 5 or so pages of code. Arrays are the most reasonable data structure to represent the neurons and weights (although linked lists can also be used, but arrays are more straightforward from programming point of view). We should parameterize the program as much as we can, so that any modification can be done easily, such as changing the number of neurons in each layer.

The number of neurons in the input and output layers is determined from the problem description. However, there is no formula for the number of hidden layers, so we have to find it from our experiment. As a gross approximation, we may start with about half as many hidden layer neurons as we have input layer neurons. When we choose too few hidden layer neurons, the computation time for each iteration will be short since the number of weights to be adjusted is small. On the other hand, the number of iterations before convergence may take a long time since there is not much freedom or flexibility for adjusting the weights to learn the required mapping. Even worse, the neural network may not be able to learn certain patterns at all. On the other hand, if we choose too many hidden layer neurons, the situation would be the opposite of too few. That is, the potential learning capability is high, but each iteration as well as the entire algorithm may be time-consuming since there are many weights to be adjusted.

We have also to find appropriate constant values of η and α from our experiment. The values of these constant affect the convergence speed, often quite significantly. The number of iterations required depends on many factors, such as the values of the constant coefficients, the number of neurons, the iteration termination criteria, the level of difficulties of sample patterns, and so on. For a simple problem like Fig. 2.4, and for relatively loose termination criteria, the number of iterations should not be excessively large; an inner loop for a single pattern may take, say, anywhere 10 to 100 iterations at the first round of training. The number will be smaller for later round of training. If the number of iterations is much higher, say, 5,000, it is likely due to an error, for example, misinterpretation of the recipe or an error in programming. For larger, real world problems, a huge number of iterations is common, sometimes taking a few weeks or even months of continuous training.

2.6 Additional Technical Remarks on the Backpropagation Model

Frequently asked questions and answers

Q. I have input and output whose values range from -100 to +800. Can I use these

raw data, or must I use normalized or scaled data, e.g., between 0 and 1, or -1 and 1?

A All these types of data representations have been used in practice.

Q. Why do we use hidden layers? Why can't we use only input and output layers?

A. Because otherwise we cannot represent mappings from input to output for many practical problems. Without a hidden layer, there is not much freedom to cope with various forms of mapping.

Q. Why do we mostly use one, rather than two or more, hidden layers?

A. The major reason is the computation time. Even with one hidden layer, often a neural network requires long training time. When the number of layers increases further, computation time often becomes prohibitive. Occasionally, however, two hidden layers are used. Sometimes for such a neural network with two hidden layers, the weights between the input and the first hidden layers are computed by using additional information, rather than backpropagation. For example, from some kind of input analysis (e.g., Fourier analysis), these weights may be estimated.

Q. The backpropagation model assumes there is a human teacher who knows what are correct answers. Why do we bother training a neural network when the answers are already known?

A. There are at least two major types of circumstances.

i) Automation of human operations. Even if the human can perform a certain task, making the computer achieve the same task makes sense since it can automate the process. For example, if the computer can identify hand-written zip code, that will automate mail sorting. Or, an experienced human expert has been performing control operations without explicit rules. In this case, the operator knows many mapping combinations from given input to output to perform the required control. When a neural network learns the set of mapping, we can automate the control.

ii) The fact that the human knows correct patterns does not necessarily mean every problem has been solved. There are many engineering, natural and social problems for which we know the circumstances and the consequences without fully understanding how they occur. For example, we may not know when a machine breaks down. Because of this, we wait until the machine breaks down then repair or replace, which may be inconvenient and costly. Suppose that we have 10,000 "patterns" of breakdown and 10,000 patterns of non-breakdown cases of this type of the machine. We may train a neural network to learn these patterns and use it for breakdown prediction. Or to predict how earthquakes or tornados occur or how the prices of stocks in the stock market behave. If we can train a neural network for pattern matching from circumstantial parameters as input to consequences as output, the results will be significant even if we still don't understand the underlying mechanism.

Q. Once a neural network has been trained successfully, it performs the required mappings as a sort of a black box. When we try to peek inside the box, we only see the many numeric values of the weights, which don't mean much to us. Can

we extract any underlying rules from the neural network?

A. This is briefly discussed at the end of this chapter. Essentially, the answer is "no."

Acceleration methods of learning

Since training a neural network for practical applications is often very time consuming, extensive research has been done to accelerate this process. The following are only few sample methods to illustrate some of the ideas.

Ordering of training patterns

As we discussed in Section 2.3, training patterns and subsequent weight modifications can be arranged in different sequences. For example, we can give the network a single pattern at a time until it is learned, update the weights, then go to the next pattern. Or we can temporarily store the new weights for several or all patterns for an epoch, then take the average for the weight adjustment. These approaches and others often result in different learning speeds. We can experiment with different approaches until we find a good one.

Another approach is to temporarily drop input patterns that yield small errors, i.e., easy patterns for the neural network for learning. Concentrate on hard patterns first, then come back to the easy ones after the hard patterns have been learned.

Dynamically controlling parameters

Rather than keeping the learning rate η and the momentum rate α as constants through out the entire iterations, select good values of these rates dynamically as iterations progress. To implement this technique, start with several predetermined values for each of η and α, for example, η = 0.4, 0.5 and 0.6, and α = 0.8, 0.9 and 1.0. Observe which pair of values give the minimum error for the first few iterations, select these values as temporary constants, as for example, η = 0.4, and α = 0.9, and perform next, say, 50 iterations. Repeat this process, that is, experiment several values for each of η and α, select new constant values, then perform next 50 iterations, and so forth.

Scaling of $\Delta w_{ij}^{(n)}$

The value of $\Delta w_{ij}^{(n)}$ determines how much change should be made on $w_{ij}^{(n)}$ to get the next iteration $w_{ij}^{(n+1)}$. Scaling up or down the value of $\Delta w_{ij}^{(n)}$ itself, in addition to the constant coefficient α, may help the iteration process depending on the circumstance. The value of $\Delta w_{ij}^{(n)}$ can be multiplied by a factor, as for example, $e^{(\rho \cos \varphi)} \cdot \Delta w_{ij}^{(n)}$, where ρ is a positive constant (e.g., 0.2) and φ is the angle between two vectors (**grad** $E(n - 1)$, **grad** $E(n)$) in the multi-dimensional space of w_{ij}'s. Here $E(n)$ is the error E at the n-th iteration. The **grad** (also often denoted by ∇) operator on scalar E gives the gradient vector of E, i.e., it represents the direction in which E increases (i.e., **-grad** represents the direction E decreases). If E is defined on only a twodimensional, xy-plane, E will be represented by the z axis. Then **grad** E would be $(\partial E/\partial x, \partial E/\partial y)$; $\partial E/\partial x$ represents the gradient in the x direction, and $\partial E/\partial y$ represents the gradient in the y direction. In our backpropagation model, **grad** E would be $(\partial E/\partial w_{11},$

$\partial E/\partial w_{12}$, ...). Note that the values of $\partial E/\partial w_{11}$, ..., have already been computed during the process of backpropagation. From analytic geometry, we have $\cos\varphi$ = **grad** $E(n$ - 1) · **grad** $E(n)$) / {|**grad** $E(n$ - 1)| |grad $E(n)$|}, where "·" is a dot product of two vectors and | | represents the length of the vector.

The meaning of this scaling is as follows. If the angle φ is 0, this means E is decreasing in the same direction in two consecutive iterations. In such a case, we would accelerate more by taking a larger value of $\Delta w_{ij}^{(n)}$. This is the case since when φ is 0, $\cos\varphi$ takes the maximum value of 1, and $e^{(\rho\cos\varphi)}$ also takes the maximum value. The value of $e^{(\rho\cos\varphi)}$ decreases when φ gets larger, and when $\varphi = 90°$, $\cos\varphi$ becomes 0, i.e., $e^{(\rho\cos\varphi)}$ becomes 1, and the factor has no scaling effect. When φ gets further larger, the value of $e^{(\rho\cos\varphi)}$ further decreases becoming less than 1, having a scaled down effect on $\Delta w_{ij}^{(n)}$. Such cautious change in $w_{ij}^{(n)}$ by the scaled down value of $\Delta w_{ij}^{(n)}$ is probably a good idea when **grad** E is wildly swinging its directions. The value of $e^{(\rho\cos\varphi)}$ becomes minimum when $\varphi = 180°$ and $\cos\varphi = -1$.

Initialization of w_{ij}

The values of the w_{ij}'s are usually initialized to uniform random numbers in a range of small numbers. In certain cases, assigning other numbers (e.g., skewed random numbers or specific values) as initial w_{ij}'s based on some sort of analysis (e.g., mathematical, statistical, or comparisons with other similar neural net works) may work better. If there are any symmetric properties among the weights, they can be incorporated throughout iterations, reducing the number of independent weights.

Application of genetic algorithms (a type of a hybrid system)

In Chapter 3 we will discuss genetic algorithms, which are computer models based on genetics and evolution. Their basic idea is to represent each solution as a collection of "genes" and make good solutions with good genes evolve, just as species evolve to better adapt to their environments. Such a technique can be applied to the learning processes of neural networks. Each neural network configuration may be characterized by a set of values such as (w_{ij}'s, θ, η, α, and possibly **x**, **y**, and **t**). Each set of these values is a solution of the genetic algorithm. We try to find (evolve) a good solution in terms of fast and stable learning. This may sound attractive, but it is a very time-consuming process.

The local minimum problem

This is a common problem whenever any gradient descent method is employed for minimization of a complex target function. The backpropagation model is not an exception to this common problem. The basic idea is illustrated in Fig. 2.10. The error E is a function of many w_{ij}'s, but only one w is considered for simplicity in this figure. E starts with a large value and decreases as iterations proceeds. If E is a smooth function, i.e., if E were a smooth monotonically decreasing curve in Fig. 2.10, E will eventually reach the global minimum, which is the real minimum. If, however, there is a bump causing a shallow valley (Local minimum 1 in the figure) so to speak, E may be trapped in this bump called a **local minimum**. E may be trapped in the local minimum since this is the only direction where E decreases in this neighborhood.

There are two problems associated to the local minima problem. One is how to detect a local minimum, and the other is how to escape once it is found. A simple practical solution is that if we find an unusually high value of E, we suspect a local minimum. To escape the local minimum, we need to "shake up" the movement of E, by applying (for example, randomized) higher values of Δw_{ij}'s.

However, we have to be cautious for the use of higher values of Δw_{ij}'s in general, including cases for escaping from a local minimum and for accelerating the iterations. If not, we may have the problem of overshooting the global minimum; even worse, we may be trapped in Local minimum 2 in the figure.

2.7 Simple Perceptrons

Typically, a backpropagation model neural network with no hidden layer, i.e., only an input layer and an output layer, is called a (simple) **perceptron**. Although practical applications of perceptrons are very limited, there are some theoretical interests on perceptrons. The reason is that theoretical analysis of practically useful neural networks is usually difficult, while that of perceptrons is easier because of their simplicity. In this section, we will discuss a few well known examples of perceptrons.

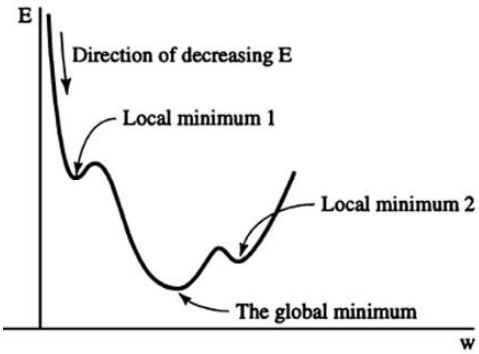


Fig. 2.10. Local minima.

Perceptron representation

Representation refers to whether a neural network is able to produce a particular function by assigning appropriate weights. How to determine these weights or whether a neural network can learn these weights is a different problem from representation. For example, a perceptron with two input neurons and one output neuron may or may not be able to represent the boolean AND function. If it can, then it may be able to learn, producing all correct answers. If it cannot, it is impossible for the neural network produce the function, no matter how the weights are adjusted. Training the perceptron for such an impossible function would be a waste of time. The perceptron learning theorem has proved that a perceptron can learn anything it can represent. We will see both types of function examples in the following, one

possible and the other impossible, by using a perceptron with two input neurons and one output neuron.

Example. x_1 AND x_2 (Boolean AND)

This example illustrates that representation of the AND function is possible. The AND function, $y = x_1$ AND x_2 should produce the following:

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

The following Fig. 2.11. shows a perceptron with two input neurons and one output neuron,which is able to represent the boolean AND function. The activation function is a step function with threshold: $y = 0$ for $net < 0.5$ and $y = 1$ otherwise.

Counter-example. x_1 XOR x_2 (Boolean XOR)

A perceptron with two input and one output cannot represent the XOR (Exclusive-OR) function, $y = x_1$ XOR x_2:

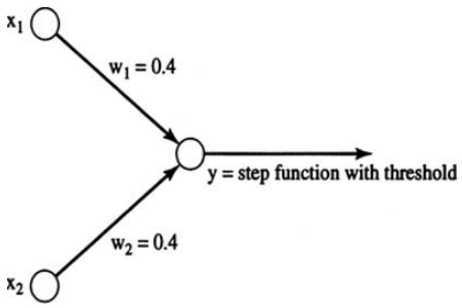


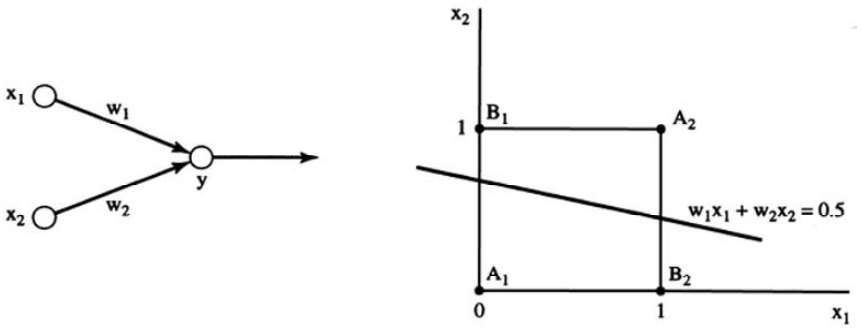
Fig. 2.11. A perceptron for the boolean AND function

x_1	x_2	y	Point in Fig. 2.12
0	0	0	A_1
0	1	1	B_1
1	0	1	B_2
1	1	0	A_2

Here Points A_1 and A_2 correspond to $y = 0$, and Points B_1 and B_2 to $y = 1$. Given a perceptron of Fig. 2.12 (a), we would like to represent the XOR function by assigning appropriate values for weights, w_1 and w_2. We will prove this is impossible no matter how we select the values of the weights.

For simplicity, let the threshold be 0.5 (we can choose threshold to be any constant; the discussion is similar. Replace 0.5 in the following with k.) Consider a

line: $w_1 x_1 + w_2 x_2 = 0.5$ (Fig. 2.12 (b)). At one side of the line, $w_1 x_1 + w_2 x_2 > 0.5$; at the other side of the line, $w_1 x_1 + w_2 x_2 < 0.5$. Changing the values of w_1, w_2, and the threshold will change the slope and position of the line. However, no line can place



(a) (b)

Fig. 2.12. Representation of the XOR function with a perceptron of (a) is impossible. No line in (b) can place A_1 and A_2 on one side and B_1 and B_2 on the other side of the line.

A_1 and A_2 on one side and B_1 and B_2 on the other side.

XOR is said to be a **linearly inseparable function**. This means that no straight line can subdivide the x_1x_2-plane to represent the function. Or, more generally, no hyperplane can subdivide an n-dimensional space to represent the function. Otherwise, when there is a line or hyperplane that can subdivide the space to represent the function, it is called a **linearly separable function**.

Example. The previous x_1 AND x_2 problem is linearly separable:

x_1	x_2	y	Point in Fig. 2.13
0	0	0	A_1
0	1	0	A_2
1	0	0	A_3
1	1	1	B_1

We can place A_1, A_2 and A_3 on one side and B_1 on the other side of the line. Hence, AND is linearly separable.

Example. The XOR problem for a neural network with a hidden layer.

This example (Fig. 2.14) demonstrates that a backpropagation model with two inputs x_1 and x_2, one hidden layer, and one output y (i.e., this is not a perceptron), can represent the XOR function as the following table. The activation functions are step functions with thresholds: $y = 0$ for $net < \theta$ and $y = 1$ otherwise. That is, the network output is 0 if the weighted sum of input $< \theta$, is 1 otherwise. The two threshold values are $\theta_1 = 1.5$ at neuron z and $\theta_2 = 0.5$ at neuron y. Fig. (a) is equivalent to Fig. (b),

which displays a layered network structure more explicitly than Fig. (a).

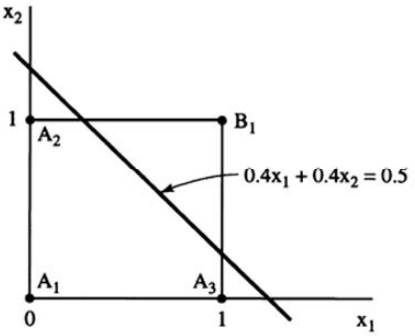


Fig. 2.13. Demonstration of a linearly separable function, AND.

x_1	x_2	*net* at *z*	Output from *z*	*net* at *y*	*y*
0	0	0	0	0	0
0	1	1	0	1	1
1	0	1	0	1	1
1	1	2	1	0	0

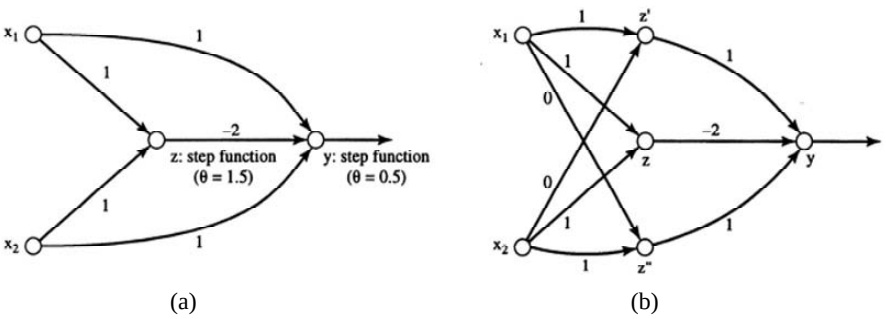


Fig. 2.14.A neural network with a hidden layer that produces the XOR function.

2.8 Applications of the Backpropagation Model

Earlier we saw an application of the backpropagation model for pattern recognition of hand-written characters (Fig. 2.4). We might tend to think such two-dimensional visual image recognition is a major application type of the backpropagation model, but actual application domains are far more versatile than this single example. The major reasons for this versatility are: 1) the input and output can have many origins rather than only visual images; 2) the functions or application types of the network can be other than recognition. Such versatility is common in many basic techniques.

For example, take a case of statistics. Statistics can be applied to problems in many disciplines: for example, social and economic data analysis such as for a census; medicine for effectiveness study of a new drug; engineering for interpretation of experimental results. Such versatility is true for the backpropagation model, other neural network models, as well as other topics discussed in this book.

Input and output types of the backpropagation model

The very basic idea of the backpropagation model is that the neural network learns *mapping* from the input of x_i's to output y_j's. In Fig. 2.4, we saw input of a two-dimensional pattern converted to one-dimensional x_i's. The following Fig. 2.15 illustrates some other forms of input. Fig. (a) shows a group of discrete input. For example, the input can be instrument measurements for an iron furnace. The first group, x_1 through x_4 are temperatures at different points in the furnace, the second group are the pressures and the third group percentages of a specific chemical component such as CO_2, at the same corresponding points as the temperatures in the furnace. As another example, the input can be some characteristic parameters for the stock market. The first group may be the Dow Jones averages for the last four days, the second group the total transaction volumes, and so forth.

Fig. (b) shows a continuous spectrum. How much fine resolution should be taken for x_i's depends on the required accuracy and computational complexity. Again, input of many applications has this form. For example, the input may be from acoustic origin rather than visual. The abscissa may represent the frequencies of sound, while the ordinate the intensity. The input may come from human speech for voice recognition, or sonar signal from the bottom of an ocean to distinguish between rock and metal (which may be an enemy submarine or a sunken ship). The input can also be from electric or electromagnetic signals such as EKG in medicine, speed/acceleration of a transportation equipment, any measurement from a machine or appliance, etc. The input can be a some measurement for economic, finance, or business decisions. In the last example, the abscissa can be time or any other attribute, while the ordinate can be a dependent parameter such as gross national product or sales amounts.

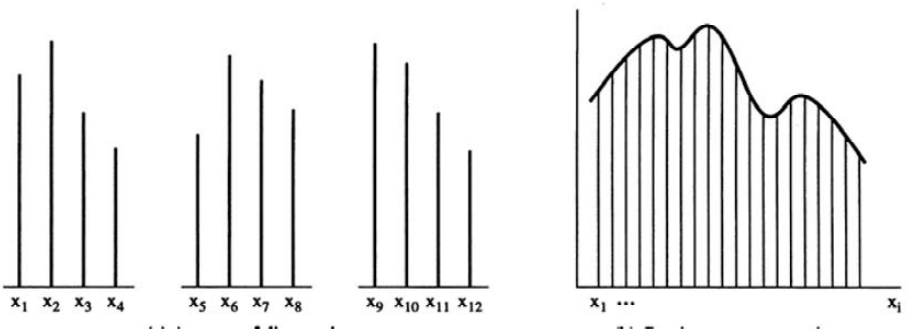


Fig. 2.15. Various forms of neural network input. (a) A group of discrete input. (b) Continuous spectrum input.

Application categories of the backpropagation model

Three major application categories of the backpropagation model are **classification**, **prediction**, and **control**. These categories are for the convenience of easy understanding, and should not be considered as exhaustive and rigid.

The character recognition example discussed in Section 2.3 to determine whether a given pattern is character *'A'* or some other character is a simple classification problem. The idea is to recognize and classify given patterns to typically much fewer groups of patterns. The latter will be the output of the network for this type of applications. This is accomplished by training the neural network using sample patterns and their correct answers. When the neural network is properly trained, it can give correct answers for not only the sample patterns, but also for new similar patterns. As discussed above, the input can be in variety of forms, not just a visual image.

When using neural networks for prediction problems, time must be incorporated as one factor. For example, suppose that a neural network is given many patterns under different circumstances over a certain length of time. Given a similar pattern in its earlier stage, the neural network may be able to predict the most likely pattern to follow. Vital issues are whether the appropriate factors are included in the model and whether they are accurately measured. For example, describing the correct financial model to predict the stock market would be difficult.

The control problem can be considered as a mapping problem from input, which may include feed-in attributes and possible feedback, to output of control parameters. The mappings of different input-to-output values can be viewed as patterns, and they can be learned by a neural network. For example, consider controlling a steel furnace. Inputs are various physical and chemical measurement distributions, such as temperature, pressure, and various chemical components at different points within the furnace. Outputs are the quantities to be supplied for the heat source, such as coal, gas and air, raw material, and so forth. Many patterns representing various input-to-output mappings can be learned by the neural network; then it can be used to control the furnace. This is an example of plant control. The same basic concept can be applied to various components of transportation equipments such as an airplane and car, robots, and so on.

2.9 General Remarks on Neural Networks

A bit of history

Although NN has a long history of research which dates back to 1943, the number of researchers involved was very small for the first 40 years. It was in 1983 that the U.S. DARPA (Defense Advanced Research Projects Agency) began funding neural network research. Soon after, other funding organizations and countries followed this initiative, and massive worldwide research began. Although there have been theoretical developments, the early optimism about practical applications of neural networks was not realized right away. For example, the learning process is difficult for many real world applications. Thus, for many years, this area created many research papers but few truly practical applications. However, this situation has

changed during the 1990s. A good number of commercial and industrial applications have been developed particularly in the U.S., Japan, and Europe.

A perspective of the neural network field as a discipline of AI

As stated in Chapter 1, There are two fundamentally different major approaches in the field of AI. One is often termed "symbolic AI," which has been historically dominant. It is characterized by a high level of abstraction and a macroscopic view. Classical psychology operates at a similar level. Classical AI areas such as knowledge engineering or expert systems, symbolic machine learning, and logic programming fall in this category. The second approach is based on low-level, microscopic biological models, similar to the emphasis in physiology or genetics. Neural networks and genetic algorithms are the prime examples of this latter approach.

The strength of neural networks is their capability to learn from patterns. This learning can then be applied to classification, prediction, or control tasks, as discussed in the previous section. Machine learning, the idea that the machine gets smarter by itself, is an extremely attractive topic. Although the history of symbolic machine learning is as old as the history of AI, its success in terms of real practicality has been relatively limited. People are excited about the neural network approach based on a different philosophy from traditional symbolic machine learning.

Although neural networks are modeled on the human brain, their current state is far from the realization of real intelligence. For one thing, we understand very little about how the brain, the model for neural networks, actually generates intelligence. There are significant differences in the physical characteristics of the brain and neural networks. The number of neurons in the brain is on the order of 10^{11}. The total number of neurons in a parallel processing system is at most on the order of 10^5, when we allocate a processor to each artificial neuron. The number of neurons in a typical neural network is much fewer, say, $O(10)$ to $O(100)$. The processing speed of a processor is, say, $O(1)$ nanoseconds or 10^{-9} seconds, while that of a natural neuron is much slower, $O(1)$ milliseconds.

Although the current neural networks are far from achieving real intelligence, they have been employed for many interesting practical applications. They include various types of pattern processing and applications to solve optimization problems. Many neural network models do not necessarily resemble their original biological counterpart, the brain.

Advantages and disadvantages of neural networks

One major advantage of neural networks is that they complement symbolic AI. For one, neural networks are based upon the brain, and for two, they are based on a totally different philosophy from symbolic AI. For this reason, neural networks have shown many interesting practical applications which are unique to neural networks.

Another major advantage of neural networks is their easy implementation of parallelism since, for example, each neuron can work independently. Generally, developing parallel algorithms for given problems or models (e.g., search, sort, matrix multiplication, etc.) is not easy. Other advantages often cited include:

Learning capability. Neural networks can learn by adjusting their weights.

Robustness. For example, neural networks can deal with certain amount of noise in the input. Even if part of a neural network is damaged (perhaps similar to partial brain damage), often it can still perform tasks to a certain extent, unlike some engineering systems, like a computer.

Generalization. A neural network can deal with new patterns which are similar to learned patterns.

Nonlinearity. Nonlinear problems are hard to solve mathematically. Neural networks can deal with any problems that can be represented as patterns.

The disadvantages of neural networks include the following: First, they have not been able to mimic the human brain or intelligence. Second, after we successfully train a neural network to perform its goal, its weights have no direct meaning to us. That is, we cannot extract any underlying rules which may be implied from the neural network. There has been some research work on this problem, but our statement is still true. A big gap remains between neural networks and symbolic AI. Perhaps this situation is essentially the same for the brain - the brain performs at a high level of intelligence, but when we examine it at the physiological level, we see only electrochemical signals passing throughout the natural neural network. A breakthrough for connecting the micro- and macroscopic phenomena in either area, artificial or natural neural networks, may solve the problem for the other. A solution for either area, however, appears unlikely to come in the near future.

Third, computation often takes a long time, and sometimes it does not even converge. A counter-argument against this common problem of long time training is that even though it may take a month of continuous training, once it is successful, it can be copied to other systems easily and the benefit can be significant. Fourth, scaling up a neural network is not a simple matter. For example, suppose that we trained a neural network for 100 input neurons. When we want to extend this to a neural network of 101 input neurons, normally we have to start over an entire training session for the new network.

Hybrid systems

One of the recent trends in research is combining neural networks and various other areas - such as fuzzy systems, genetic algorithms and expert systems - to create hybrid systems. The fundamental concept behind such hybrid systems is for each component to complement each other's weakness, thus creating new approaches to solve problems. For example, there are no capabilities for machine learning in fuzzy systems. Fuzzy systems do not have capabilities of memorizing and recognizing patterns in the way neural networks do. Fuzzy systems with neural networks may add these capabilities. A neural network may be added to an expert system as a front or back end, analyzing or creating patterns.

Further Reading

The two volumes by Anderson, et al. include many reprinted seminal articles. The two volumes by Rumelhart/McClelland are very frequently cited references.

J.A. Anderson and E. Rosenfeld (Eds.), *Neurocomputing: Foundations of Research*, MIT Press, 1988.

J.A. Anderson, A. Pellionisz and E. Rosenfeld (Eds.), *Neurocomputing 2: Directions for Research*, MIT Press, 1990.

L. Fausett, *Fundamentals of Neural Networks*, Prentice-Hall, 1994.

R. Hecht-Nielsen, *Neurocomputing*, Addison-Wesley, 1990.

J. Hertz, A. Krogh and R.G. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, 1991.

S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd Ed., Prentice-Hall, 1999.

B. Müller, J. Reinhardt and M.T. Strickland, *Neural Networks: An Introduction*, 2nd Ed., Springer, 2002.

D.E. Rumelhart, J.L. McClelland and the PDP Research Group (Eds.), *Parallel Distributed Processing*, Vols. 1 and 2, MIT Press, 1986 (especially Vol. 1, Chapter 8 for the backpropagation model).

There are many journals that carry articles in the field including the following.

IEEE Transactions on Neural Networks.
Neural Networks, Elsevier Science.

Also many, often very thick, conference proceedings have been published by several organizations.

3 Neural Networks: Other Models

3.1 Prelude

There are many other models of neural networks in addition to the backpropagation model discussed in the previous chapter. Although the backpropagation model is the one most widely employed for practical use, other models have interesting applications. In this chapter we will discuss major characteristics that can be associated to neural networks and several selected other models.

Characteristics that derive different neural network models

As stated before, the basic building block of every neural network model is a neuron, described already in Section 2.2. Although we start with the same building blocks, there are many neural network models. The reason is that there are various characteristics associated with the neural networks, and by selecting different characteristic values, we can come up many different models. As an analogy, we can make different foods using the same materials but by performing different methods of cutting, seasoning, and cooking. It is sometimes overwhelming to understand exact definitions of many models and their variations, especially at the beginning. One reasonable approach may be to study a few well-known selected models, understand the nature of their associated characteristics, and extend the concepts learned to further additional models whenever necessary.

The following list includes some of the characteristics associated with various types of neural networks and their typical meanings. The meanings of some of the characteristics will become clearer when we study specific models with these characteristics. Often, there are many variations of these networks and the following describes the most common scenarios.

Multilayered/non-multilayered - Topology of the network architecture

Multilayered
The backpropagation model is multilayered since it has distinct layers such as input, hidden, and output. The neurons within each layer are connected with the neurons of the adjacent layers through directed edges. There are no connections among the neurons within the same layer.

Non-multilayered
We can also build neural network without such distinct layers as input, output, or hidden. Every neuron can be connected with every other neuron in the network through directed edges. Every neuron may input as well as output. A typical example is the Hopfield model.

Non-recurrent/recurrent - Directions of output

Non-recurrent (feedforward only)
In the backpropagation model, the outputs always propagate from left to right in the diagrams. This type of output propagation is called *feedforward*. In this type, outputs from the input layer neurons propagate to the right, becoming inputs to the hidden layer neurons, and then outputs from the hidden layer neurons propagate to the right becoming inputs to the output layer neurons. Neural network models with feedforward only are called **non-recurrent**. Incidentally, "backpropagation" in the backpropagation model should not be confused with feedbackward. The backpropagation is backward adjustments of the weights, not output movements from neurons.

Recurrent (both feedforward and feedbackward)
In some other neural network models, outputs can also propagate backward, i.e., from right to left. This is called **feedbackward**. A neural network in which the outputs can propagate in both directions, forward and backward, is called a **recurrent model**. Biological systems have such recurrent structures. A feedback system can be represented by an equivalent feedforward system (see Rumelhart, et al., 1986, Vol. 1, p. 355).

Supervised/unsupervised learning - Form of learning

Supervised learning
For each input, a teacher knows what should be the correct output and this information is given to the neural network. This is **supervised learning** since the neural network learns under supervision of the teacher. The backpropagation model is such an example, assuming an existence of a teacher who knows what are correct patterns. In the backpropagation model, the actual output from the neural network is compared with the correct one, and the weights are adjusted to reduce the difference.

Unsupervised learning
In some models, neural networks can learn by themselves after being given some form of general guidelines. There is no external comparison between actual and ideal output. Instead, the neural network adjusts by itself internally using certain criteria or algorithms - e.g., to minimize a function (e.g., "global energy") defined on the neural network. Such form of learning is called **unsupervised learning**. (Unsupervised learning does not mean no guidance is given to the neural network; if no direction is given, the neural network will do nothing.)

Binary/bipolar/continuous input - Types of input values

We can assume different types of input values. The most common types are binary (i.e., an input value is restricted to either 0 or 1), bipolar (an input value is either -1 or 1), and continuous (i.e., continuous real numbers in a certain range).

Activation functions of linear, step, sigmoid, etc. - Forms of activation functions

Various activation functions can be used, including those discussed in Section 2.2.

We will see examples of these various characteristics as we discuss other neural network models in this chapter. Some well known neural network models and their associated typical characteristics are:

Backpropagation:	multilayered, nonrecurrent, supervised
Hopfield:	non-multilayered, recurrent, supervised
Kohonen:	multilayered, nonrecurrent, unsupervised
Boltzmann machine:	non-multilayered, recurrent, supervised/unsupervised

Organization of neural network models

We can organize neural network models based on these different criteria. The following are based on the two most common criteria, with over-simplified entries for simplicity.

1. Based on the *functional characteristics*

Functional Characteristics	Representative Model
Pattern classification	Backpropagation
Associative memory	Hopfield
Optimization	Hopfield-Tank
Clustering	Kohonen
Clustering, optimization	Boltzmann machine

2. Based on *specific models*, often named after the original developer

Model	Functional Characteristics
Backpropagation	Pattern classification
Hopfield	Associative memory
Hopfield-Tank	Optimization
Kohonen	Clustering
Boltzmann machine	Clustering, optimization

Both ways of organizations are used in the literature. Some authors may prefer the first organization since it addresses what types of functions or tasks the neural network models are aimed to perform. Although there is only one entry in each functional characteristic listed above, often there is more than one model aimed to

perform the same type of task. Grouping models for the same tasks would be convenient to compare alternative approaches to tackle the problem.

Historically, however, the second type of organization preceded the first type. These models were developed by certain researchers, and people referred to these models by the developers' names; thus the names were and still are popular. It is probably easier to remember and identify these models by their names rather than by their functional characteristics. In this book, we will basically use the second method of organization, since the objective here is to introduce the basics rather than to provide an extensive coverage of the field. With this introductory background, we will be able to understand extensions of the basics whenever necessary.

3.2 Associative Memory

In *Webster's New World Dictionary*, "association" is described as "a connection in the mind between ideas, sensations, memories, etc." The human brain can associate different types of inputs; for example, we can associate a visual appearance and voice with a specific person. We can also associate modified values of the same type of inputs. We can recognize a picture of a person, for instance, or the face of a friend we have not seen for ten years.

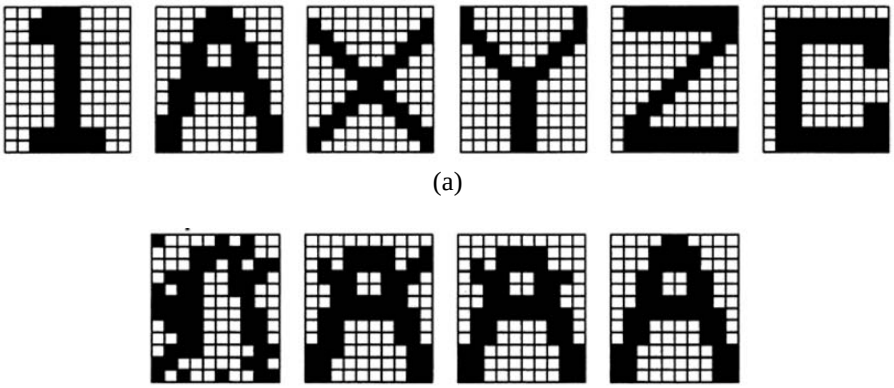
Ordinary computer memory is non-associative. That is, an exact memory *address* must be specified, and the only information at this address is retrieved. Certain types of neural networks can be used as **associative memory** (or **content-addressable memory**). Associative memory is a type of neural network that can map (associate) inputs, which are *contents* rather than addresses, to information stored in memory. That is, given an input (which may be partial, noisy, or may contain an error), the network can retrieve a complete memory which is the closest match to the input. Mathematically, this property can be stated as mapping an input vector $\mathbf{x}$ to the closest vector $\mathbf{x}^{(s)}$ among vectors $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, ..., $\mathbf{x}^{(m)}$. To implement an associative memory, we can use a recurrent neural network and select appropriate weights in such a way that desired stable outputs will come out for given inputs.

For example, suppose "H.A. Kramers & G.H. Wannier, Phys. Rev. 60, 252 (1941)" is stored in the computer memory. By giving sufficient partial information, such as "& Wannier, (1941)", our associative memory would be capable of retrieving the entire memory. An ideal memory could deal with errors and retrieve this memory even from input "Vannier, (1941)" [Hopfield, 1982]. The following is another simple example of associative memory.

Example. Pattern association.

Fig. 3.1(a) shows six exemplar patterns of the characters "1," "A," "X," "Y," "Z," and "C." Each pattern is drawn on a 10×12 pixel board, which can be converted to a one-dimensional array or a vector of $10 \times 12 = 120$ components. Note that the two-dimensional view is only for human perception; the computer does not understand these patterns as two-dimensional. The value of each vector element is -1 if the corresponding pixel on the board is white (blank), 1 if black. Fig. 3.1(b)

shows how a noisy input pattern converges to the closest exemplar pattern during iterations. These figures were drawn by my graduate student Bill Leach. The $m = 6$ exemplar patterns can be denoted as vectors $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, ..., and $\mathbf{x}^{(6)}$, and the noisy input vector as $\mathbf{x}$. This type of pattern association can be implemented by using associative memory, such as the Hopfield network discussed in the next section.



(a)

(b)

Fig. 3.1. Demonstration of associative memory by pattern association. (a) Six exemplar patterns. (b) A noisy input pattern approaches its closest exemplar pattern.

Many types of associative memories have been proposed. In this chapter, we illustrate the basic idea by the Hopfield network.

3.3 Hopfield Networks

The Hopfield network model is probably the second most popular type of neural network after the backpropagation model. There are several versions of Hopfield networks. They can be used as *associative memory*, as we will discuss in this section, and they can also be applied to *optimization* problems, as we will study in the next section. The version for the associative memory is classified as supervised learning by some authors and as unsupervised by others, with the distinction based on the authors' interpretation of the definitions. Given that the network performs pattern association under the supervision of a teacher, we use the former definition in this book.

The basic idea of the Hopfield network is that it can store a set of exemplar patterns as multiple stable states. Given a new input pattern, which may be partial or noisy, the network can converge to one of the exemplar patterns that is nearest to the input pattern. This is the basic concept of applying the Hopfield network as associative memory.

Architecture

As shown in Fig. 3.2, a Hopfield network consists of a single layer of neurons, 1, 2, ..., *n*. The network is **fully interconnected**; that is, every neuron in the network is connected to every other neuron. The network is **recurrent**; that is, it has **feedforward/feedbackward** capabilities, which means input to the neurons comes from external input as well as from the neurons themselves internally.

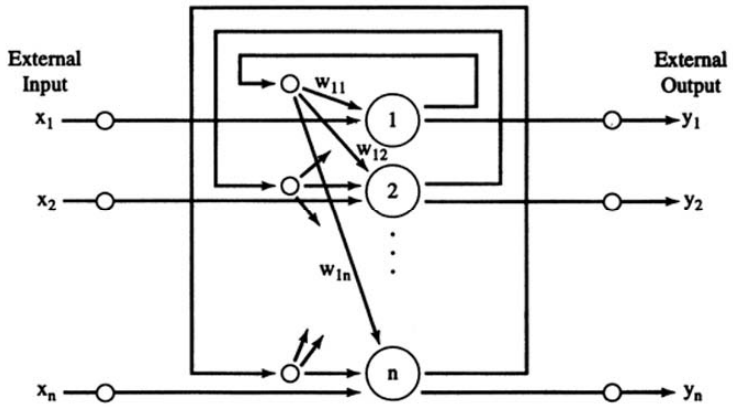


Fig. 3.2. A Hopfield network configuration.

Each input/output, x_i or y_j, takes a discrete bipolar value of either 1 or -1. The number of neurons, *n*, is the size required for each pattern in the bipolar representation. For example, suppose that each pattern is a letter represented by an 10×12 two-dimensional array, where each array element is either 1 for a black square or -1 for a blank square (for example, Fig. 3.1). Then *n* will be $10 \times 12 = 120$.

Each edge is associated by weight, w_{ij}, which satisfies the following conditions:

$$w_{ij} = w_{ji} \text{ for all } i, j = 1, n$$

and

$$w_{ii} = 0 \text{ for all } i = 1, n.$$

(Because $w_{ii} = 0$, the self-coupling edge of Neuron *i*, i.e., the edge from Neuron *i* to Neuron *i*, can be considered as "not connected.") The values of w_{ij} are specified in advance from exemplar patterns and fixed as we will see in the following example.

Computational procedures

Determining w_{ij}

Suppose that m exemplar patterns are presented ($s = 1, m$). Each pattern $\mathbf{x}^{(s)}$ has *n* inputs, $x_1^{(s)}, x_2^{(s)}, ..., x_n^{(s)}$, where $x_k^{(s)} = 1$ or -1. Determine w_{ij} for $i, j = 1$ to *n* by:

$$w_{ij} = \begin{cases} 0 & \text{for } i = j \\ \sum_{s=1}^{m} x_i^{(s)} x_j^{(s)} & \text{for } i \neq j \end{cases}$$

After this determination of w_{ij}, the values of w_{ij}'s are not changed. This feature is different from the backpropagation model, where w_{ij}'s are changed as the learning process proceeds. The Hopfield network is classified under supervised learning since at the beginning it is given correct exemplar patterns by a teacher.

Interpretation of w_{ij} is as follows. For a given pattern s, if $x_i^{(s)} = 1$ and $x_j^{(s)} = 1$ (i.e., both neurons i and j are active), or if $x_i^{(s)} = -1$ and $x_j^{(s)} = -1$ (i.e., both neurons are inactive), then $x_i^{(s)} x_j^{(s)} = 1$ and a positive contribution to w_{ij} results. If this occurs for the majority of m patterns, then $w_{ij} > 0$, i.e., the synapse between neurons i and j becomes **excitatory**. The higher the number of such patterns, the more excitatory the synapse.

On the other hand, if $x_i^{(s)} = 1$ and $x_j^{(s)} = -1$, or if $x_i^{(s)} = -1$ and $x_j^{(s)} = 1$ (i.e., if $x_i^{(s)} x_j^{(s)} = -1$), then a negative contribution to w_{ij} results. If this occurs for the majority of patterns, then $w_{ij} < 0$, i.e., the synapse becomes **inhibitory**.

New input $x_i(0)$, and x_i

Every input $x_i(0)$ and x_i is bipolar (i.e., either -1 or +1). The initial values of $x_i(0)$, $i = 1, n$ are given at time $t = 0$. Let $net_i(t) = \Sigma^n_{j=1} w_{ij} x_j(t)$. Then $x_i(t+1)$, for $i = 1$ to n is determined as follows:

$$x_i(t+1) = \begin{cases} 1 & \text{if } net_i(t) > \theta_i \\ x_i(t) & \text{if } net_i(t) = \theta_i \\ -1 & \text{if } net_i(t) < \theta_i \end{cases}$$

Here θ_i are the thresholds. θ_i are usually set to 0.

The neurons are updated one at a time for a specific value of i by $x_i(t+1) \leftarrow x_i(t)$. The only constraint on the updates is that all the neurons must be updated at the same average rate. Often the neurons are picked out at a uniformly random rate. Starting with a given initial input $x_i(0)$, $x_i(t)$ can converge to the closest stored pattern.

An intuitive interpretation of this process follows. Assume that we have a small number of uncorrelated exemplar patterns in comparison with the number of neurons. net_i in the above formula at iteration step t can be expressed by using the following definitions for net_i and w_{ij}:

$$\begin{aligned} net_i &= \sum_{j=1}^{n} w_{ij} x_j \\ &= \sum_{j=1, j\neq i}^{n} \left(\sum_{s=1}^{m} x_i^{(s)} x_j^{(s)} \right) x_j \\ &= \sum_{s=1}^{m} x_i^{(s)} \sum_{j=1, j\neq i}^{n} x_j^{(s)} x_j \end{aligned}$$

At the last step, we swapped the two summations (one for *s* and the other for *j*) since they are independent, and factored $x_i^{(s)}$, which does not depend on *j*, outside of $\Sigma^n_{j=1,\, j\neq i}$.

Now imagine that the unknown input pattern **x** closely resembles a specific exemplar pattern $\mathbf{x}^{(s')}$ and is totally uncorrelated to the remaining exemplar patterns. The last factor $\Sigma^n_{j=1,\, j\neq i}\, x_j^{(s)}\, x_j$ will have a large value for the close exemplar pattern since $x^{(s')}\, x_j$ will be 1 for most of *j* (this is because when x_j is 1, $x^{(s')}$ is likely to be 1, and when x_j is -1, $x_j^{(s')}$ is likely to be -1). If x_j and $x_j^{(s')}$ completely match for all of j except $j = i$, the summation will be n - 1. The first factor, $x^{(s')}$ is multiplied by the large second factor and will significantly contribute toward net_i. On the other hand, the second factor, $\Sigma^n_{j=1,\, j\neq i}\, x^{(s)}\, x_j$, will be much smaller for the remaining uncorrelated exemplar patterns. This is because sometimes both x_j and $x_j^{(s')}$ have the same value (either 1 or -1), resulting in $x_j^{(s)}\, x_j = 1$; some other times x_j and $x_j^{(s')}$ have opposite values, resulting in $x_j^{(s)}\, x_j = -1$; thus, the 1's and -1's cancel out, yielding a small sum. This implies that the contributions from the remaining patterns toward net_i are small. The overall effect is that $\mathbf{x}(t+1)$ of the next iteration step will be closer to $\mathbf{x}^{(s')}$, or $x_i(t+1)$ will be closer to $x_i^{(s')}$ in terms of each component.

The actual outcome of the converged solution may not necessarily be the closest matched exemplar pattern. It can be some other exemplar pattern or a pattern different from any of the exemplar patterns. To reduce the probability of such an error, the number of exemplar patterns, *m*, should be less than $0.15n$ [Hopfield, 1982]. In practice, *m* is typically kept well below $0.15n$. Several factors affect the outcome of a converged solution. They include the ratio m/n, correlation among the exemplar patterns, the initial values of $x_i(0)$'s, and the updating processes (the scheme of picking out the neurons and the random number seeds).

Energy function

Define *E*, an **energy** (or **Lyapunov**) **function**, as:

$$E = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}x_i(t)x_j(t) + \sum_{i=1}^{n}\theta_i x_i(t)$$

We can prove that as iterations proceed, *E* always decreases when $x_i(t)$ changes the value, and *E* stays the same when there is no change in $x_i(t)$'s.

When we think of the term "energy," we think of a physical quantity such as kinetic or electric energy. This was probably the idea when the equation was originally defined, but we interpret our energy in much a broader sense in our applications. The term "energy" here represents a measure that reflects the state of the solution, and is somewhat analogous to the concept of physical energy.

Basic processing steps

Step 1. Storing exemplar patterns as preprocessing.

Determine w_{ij} for $i, j = 1, n$, for exemplar patterns using the "Determining w_{ij}" procedure described above.

Step 2. Finding the closest matching exemplar pattern to given input representing an unknown pattern.

At $t = 0$, apply given input $x_i(0)$, $i = 1, n$.

Perform iterations updating $x_i(t)$'s until the energy function E stops decreasing (or, equivalently, $x_i(t)$'s remain unchanged). Then $x_i(t)$ represents the solution, that is, the exemplar pattern that best matches (*associates* to) the unknown input. The converged $x_i(t)$ can be sent out externally as output y_i.

Implementation considerations of Step 2 above

There are two conditions:

1. Selection of neurons. Each neuron is picked out at uniformly random, independent of other neurons, and at the same average rate. For each neuron x_i, the new value updates the value of x_i, and will be used in the computation of other neurons (and x_i itself, if it is picked up again).
2. Convergence. If and only if a neuron changes its state, then the energy decreases. A solution is converged upon if all the neurons are updated without any change.

The following are possible methods for examining the above conditions.

1. Select $i = 1, 2, ..., n$, in this order (this is an epoch). Test for convergence. If not converged, then go to the next epoch. This method violates Condition 1 (it is not random).
2. Select i from 1 to n, at a uniformly random rate, independent of other neurons, n times (this is an epoch). Test for convergence. If not converged, then go to the next epoch. This method violates Condition 2 (some neurons may not be updated).
3. Select a unique i every time from 1 to n, in random order, n times; that is, every number between 1 to n is picked up once and only once (this is an epoch). Test for convergence. If not converged, then go to the next epoch. This method can be implemented by randomly permuting numbers 1 to n, then picking out one number at a time in order. This method violates Condition 1 (neurons are not picked out independently of other neurons, because the probabilities of neurons being picked out gets higher when they have not been picked out).
4. Select i from 1 to n, at a uniformly random rate, independent of other neurons, until every neuron is updated at least once (this is an epoch). Test for convergence. If not converged, then go to the next epoch.

A possible implementation of Method 4 is: Define an "update vector," $\mathbf{q} = (q_1, \ldots, q_n)$, where $q_i = 1$ if x_i has been updated at least once since the beginning of the epoch, otherwise $q_i = 0$. For each epoch,
Initialize: $\mathbf{q} = \mathbf{0}$, i.e., $q_i = 0$ for $i = 1, n$. count = n, where count is the number of non-updated neurons.
Repeat: Every time x_i is updated, check q_i. If $q_i = 0$, then set $q_i = 1$ and $c = c - 1$; if $c = 0$, then the end of the epoch is reached.
This last method has no violation of either Condition 1 or Condition 2. Methods 1, 2

and 3 involve Condition 1 or 2 violation, but they seem to be used in practice. My students experimented with Methods 1 through 4. For midsize problems (say, $n = 120$), Method 4 took about 5 to 7 times the number of iterations of the other three for an epoch. The number of epochs required for the network to converge was about the same for all the methods.

Example.

The pattern association example (Fig. 3.1) can be implemented by the Hopfield network architecture and algorithm discussed in this section. The network has 120 neurons. In Step 1, the network stores $m = 6$ exemplar patterns given in Fig. 3.1(a), by assigning appropriate values of w_{ij}. At $t = 0$ in Step 2, the network is given the unknown pattern (the first pattern in Fig. 3.1(b), x_i, $i = 1, 120$). During the subsequent iterations, $x_i(t)$ will gradually converge to the exemplar pattern"A" that matches the unknown input.

3.4 The Hopfield-Tank Model for Optimization Problems: The Basics

We will now discuss a neural network model which is particularly popular in optimization problems. In their 1985 article, Hopfield and Tank reported that the traveling salesman problem (TSP) can be solved much faster by using their model than by using existing methods. Although the model does not necessarily determine the optimal solution, it can find solutions which are close enough for practical purposes. This was a blockbuster for the computing community, since solving NP-complete problems in an efficient way has been a major obstacle. Soon after the publication of the article, several papers came out which contested Hopfield and Tank's claims. Since then, however, more and more application problems have been solved using the model, and it has become a major optimization technique. Since optimization problems are very common and important in many disciplines, such as engineering and management, this model will be a valuable alternative to other classical techniques found in calculus and operations research.

In this section, we will discuss the basics of the Hopfield-Tank technique; in the next section, we will study some application examples and provide a brief general guideline to apply the model to optimization problems. We will describe the model in a one-dimensional case since it is fundamental and easy to understand. We will then extend it to a two-dimensional configuration, since for certain problems this approach is more efficient. Extensions for higher dimensions can be done similarly.

3.4.1 One-Dimensional Layout

Imagine that n neurons, $i = 1$ to n, are one-dimensionally laid out. Weight w_{ij} is associated with the edge from neuron i to neuron j. Assume the symmetric property for the weights, $w_{ij} = w_{ji}$.

Computational procedures

Basic equations

The equations of motion are given as follows:

$$\frac{du_i}{dt} = -u_i + \sum_{j=1}^{n} w_{ij}V_j + I_i \qquad \text{for } i = 1, n \tag{1}$$

$$u_i^{(t+1)} = u_i^{(t)} + \frac{du_i}{dt} \cdot \Delta t \qquad \text{for } i = 1, n \tag{2}$$

$$V_j = g(u_j) = \frac{1}{2}\left\{1 + \tanh\frac{u_j}{u_0}\right\} \qquad \text{for } j = 1, n \tag{3}$$

The energy function is defined as follows:

$$E = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}V_iV_j - \sum_{i=1}^{n} V_iI_i \tag{4}$$

In the above, u_i is the net input, V_i is the output, and I_i is the external input for neuron i; w_{ij}, I_i, for $i, j = 1, n$, and u_0 are constant values determined appropriately by the problem characteristics. In equation (2), $u_i^{(t)}$ represents u_i at time step t, and Δt is a small constant representing a time increment on each step. Equation (3) for V_j is a sigmoid function between 0 and 1 (Fig. 3.3). Note that the smaller the value of u_0, the steeper the slope of the graph. Recall that $\tanh x = (e^x - e^{-x})/(e^x + e^{-x})$. The problem is to determine V_i for $i = 1, n$ in such a way that energy E in equation (4) becomes minimum.

As mentioned in the previous section, the term "energy" should be interpreted in a broader sense rather than physical energy. For example, in optimization problems, our energy can be a cost, time, or distance to be minimized. Similarly, by the equations of motion, we might imagine physical kinetic equations for a hard ball rolling on a frictionless hard surface. Rather, theses equations should be interpreted in a much broader sense in applications; the basic idea is that these equations force our solution to the desired direction.

Iteration process to determine V_i for $i = 1, n$

Initialization

Step 0. Choose u_i at $t = 0$ (may be denoted as $u_i^{(0)}$) at random; for example,

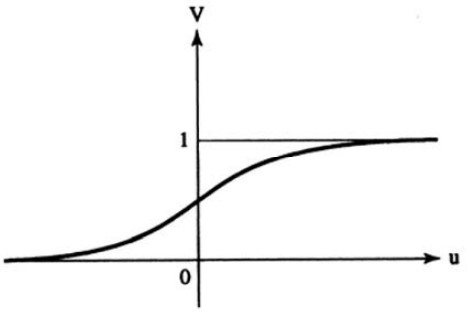


Fig. 3.3. A sigmoid function for V in terms of u.

$$u_i^{(0)} = u_0 + \delta u_i$$

where δu_i is a random number chosen uniformly in the interval of -0.1 $u_0 \le \delta u_i \le 0.1\ u_0$. Also, choose a large positive number as a dummy initial value of E.

Iteration. Repeat the following steps until E stops decreasing.

Step 1. Compute V_j from u_j by equation (3) for $j = 1, n$.

Step 2. Compute E using equation (4).

Step 3. Compare the current $E^{(t)}$ with $E^{(t-1)}$, the E value one iteration before.
(i) If $E^{(t)} \ge E^{(t-1)}$, stop iteration; V_j, for $j = 1, n$ are the *solutions*.
(ii) Otherwise continue (go to Step 4).

Step 4. Using equation (1) compute du_i/dt for $i = 1, n$. Using equation (2), compute u_i for $i = 1, n$ for the next iteration step. (Go to Step 1.)

3.4.2 Two-Dimensional Layout

There are many interesting application problems that can be associated with an $n \times n$ square matrix. For these problems, representing the previously discussed variables and expressions in two-dimension form is convenient. For easy understanding, we discuss one- and two-dimensional configurations in two steps. The major difference in two-dimensional configuration is the way the variable subscripts are represented. The variables with a single subscript in the one-dimensional case will be replaced with variables with double subscripts.

An alternative method to the two-dimensional approach covered in this subsection is to "spread out" the two-dimensional elements into one-dimensional arrangement and use the previous procedures. For an $n \times n$ two-dimensional array, the spread-out subscripts will be: 1, 2, ..., n, $n + 1$, ..., n^2. The idea can be extended to higher-dimensions.

Imagine that n^2 neurons are laid out two-dimensionally, in the same configuration as the elements in an $n \times n$ square matrix. The neurons are identified by row and column numbers; Neuron *ik* is the *i*-th row, *k*-th column unit. The entire set of neurons can be represented as Neuron *ik*, for $i = 1, n$ and $k = 1, n$. The weight associated with the edge from neuron *ik* to neuron $j\ell$ is denoted as $w_{ik,j\ell}$. We can obtain the basic equations for two-dimensional layout by replacing the subscripts and summations in the one-dimensional case as: *i* by *ik*, *j* by $j\ell$, Σ_i by $\Sigma_i \Sigma_k$, and Σ_j by $\Sigma_j \Sigma_\ell$.

Computational procedures

Basic equations

The equations of motion are given as follows:

$$\frac{du_{ik}}{dt} = -u_{ik} + \sum_j \sum_\ell w_{ik,\,j\ell} V_{j\ell} + I_{ik} \qquad \text{for } i = 1, n \text{ and } k = 1, n \qquad (1')$$

$$u_{ik}^{(t+1)} = u_{ik}^{(t)} + \frac{du_{ik}}{dt} \cdot \triangle t \qquad \text{for } i = 1, n \text{ and } k = 1, n \qquad (2')$$

$$V_{j\ell} = g(u_{j\ell}) = \frac{1}{2}\left\{1 + \tanh \frac{u_{j\ell}}{u_0}\right\} \qquad \text{for } j = 1, n \text{ and } \ell = 1, n \qquad (3')$$

The energy function is defined as:

$$E = -\frac{1}{2} \sum_i \sum_k \sum_j \sum_\ell w_{ik,\,j\ell} V_{ik} V_{j\ell} - \sum_i \sum_k V_{ik} I_{ik} \qquad (4')$$

The iteration process is the same as before except that n^2 (instead of n) values of u_{ik} and V_{ik} are computed.

3.5 The Hopfield-Tank Model for Optimization Problems: Applications

3.5.1 The *n*-Queen Problem

We now consider a simple example to illustrate how the basic equations are set up and how iterations proceed for a specific problem. Our simple example is the *n*-queen problem, which happens to be NP-complete (which means computationally hard).

Probably most of us are already familiar with the *n*-queen problem. On an $n \times n$ chessboard, a queen can move any number of squares up, down, to the right, to the left, and diagonally. Fig. 3.4 (a) illustrates a possible move for a queen for a case of $n = 5$. The problem is to determine *n* "safe" positions for *n* queens. A safe position means that none of the queens can move to a square occupied by another queen in

only one move. Obviously there must be one and only one queen in each row and column. (If two or more queens are in a row or column, then one would be attacked by another; if no queens are in a row or column, there must be a row or column that has more than one queen.) Similarly, there must be at most one queen in each diagonal direction (since there can be no queen in a diagonal direction). Fig. 3.4 (b) is a solution for $n = 5$; generally, a solution is not unique for a specific value of n.

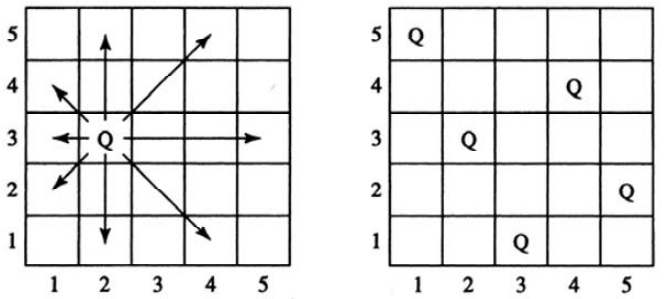


Fig. 3.4. The n-queen problem illustration for $n = 5$. (a) A possible move of a queen. (b) A solution.

Formulation of the problem

Conditions to be satisfied

1. Exactly one queen in each *row*.
2. Exactly one queen in each *column*.
3. At most one queen in *upward diagonal* (from left-bottom to right-top) (Fig. 3.5(a)).
4. At most one queen in *downward diagonal* (from left-top to right-bottom) (Fig. 3.5 (b)).

Diagonal position representations

Given a square position (i, j) on the chessboard, its diagonal positions can be represented as follows (Figs. 3.4 and 3.5).

Upward diagonal: $(i+k, j+k)$, where $k = \max [1-i, 1-j]$ to $\min [n-i, n-j]$.

Downward diagonal: $(i+k, j-k)$, where $k = \max [1-i, j-n]$ to $\min [n-i, j-1]$.

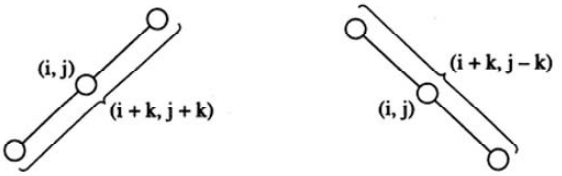


Fig. 3.5 (a) Upward diagonal (from left-bottom to right-top) squares. (b) Downward diagonal (from left-top to right-bottom) squares.

The following Fig. 3.6 (a) and (b) illustrate two examples for upward diagonal positions.

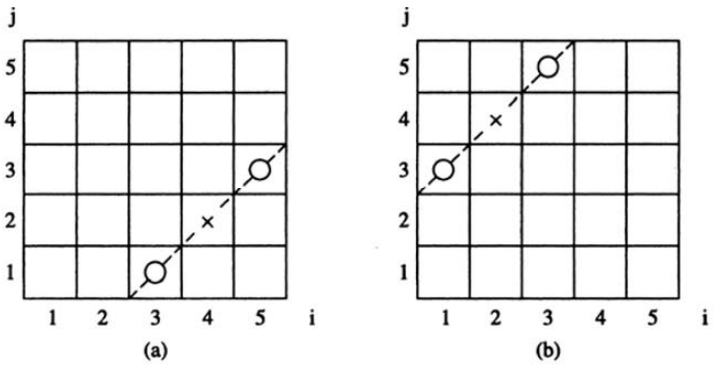


Fig. 3.6.Two examples of upward diagonal positions.

In Fig. 3.6 (a), (i, j) is (4, 2), and k = -1, 0, and 1 for $(i + k, j + k)$ will give the three diagonal positions, (3, 1), (4, 2) and (5, 3), respectively. Similarly, in Fig. 3.6 (b), (i, j) is (2, 4), and k = -1, 0, and 1 for $(i + k, j + k)$ will represent the three diagonal positions, (1, 3), (2, 4) and (3, 5), respectively. In general, the range of k for the upward diagonal must satisfy $(1 \le i + k \le n)$ *and* $(1 \le j + k \le n)$ to stay inside of the range of the chessboard. By moving i and j to the outer expressions, we have $(1 - i \le k \le n - i)$ and $(1 - j \le k \le n - j)$. For the lower bound of k, k must satisfy both $1 - i \le k$ and $1 - j \le k$, which means k must be greater than or equal to whichever the larger of $1 - i$ and $1 - j$. Hence, the lower bound of k is $\max[1 - i, 1 - j]$. Similarly, the upper bound of k is $\min[n - i, n - j]$.

For downward diagonal, k must satisfy $(1 \le i + k \le n)$ and $(1 \le j - k \le n)$ to stay inside of the chessboard. This leads to the range of k as $\max [1 - i, j - n]$ to $\min [n - i, j - 1]$.

Definition of V_{ij}

$$V_{ij} = \begin{cases} 0 & \text{if no queen on } ij\text{-location} \\ 1 & \text{otherwise} \end{cases}$$

Since V_{ij} represents a solution, and our solution should give either "yes queen" or "no queen" for each square, the above is a reasonable definition for V_{ij}.

Basic equations

Now we define our equations in such a way to drive our solution in a better and better direction in terms of satisfying the conditions. The equation of motion for du_{ij}/dt can be defined either to increase or decrease u_{ij}, which in turn either to increase or decrease V_{ij}. To increase u_{ij}, we should make $du_{ij}/dt > 0$; to decrease u_{ij}, we should make $du_{ij}/dt < 0$; in case we want to keep the current value of u_{ij}, we should make $du_{ij}/dt = 0$. With these considerations in mind, we propose the following equation of motion.

$$\frac{du_{ij}}{dt} = -\left\{ \left(\sum_{k=1}^{n} V_{ik} - 1 \right) + \left(\sum_{k=1}^{n} V_{kj} - 1 \right) + \left(\sum_{\substack{k=\max[1-i,1-j] \\ \text{to}\min[n-i,n-j] \\ \text{and}(k\neq 0)}} V_{i+k,\,j+k} \right) + \left(\sum_{\substack{k=\max[1-i,\,j-n] \\ \text{to}\min[n-i,\,j-1] \\ \text{and}(k\neq 0)}} V_{i+k,\,j-k} \right) \right\}$$

On the right-hand side of the equation, we have four terms. Let us examine the effect of the first term: ($\Sigma^{n}_{k=1} V_{ik}$ - 1) (which is inside of -{...}). Our definition of V_{ij} is: V_{ij} = 0 if no queen, at square *ij*, V_{ij} = 1 otherwise. Thus, the sum $\Sigma^{n}_{k=1} V_{ik}$ adds up the values of *V* in column *i*. If there is one queen in column *i* (which is ideal for the column), $\Sigma^{n}_{k=1} V_{ik}$ will be 1, and the first term ($\Sigma^{n}_{k=1} V_{ik}$ - 1) will be zero; i.e., no first-term effect to du_{ij}/dt, or change in u_{ij} caused by the first term. If there is no queen in column *i*, $\Sigma^{n}_{k=1} V_{ik}$ will be 0, and the first term ($\Sigma^{n}_{k=1} V_{ik}$ - 1) will be negative, or more precisely, -1. Then - { ($\Sigma^{n}_{k=1} V_{ik}$ - 1)} will be positive, and this term will contribute to $du_{ij}/dt > 0$, i.e., it will increase u_{ij}. The effect is to increase V_{ij}, the number of queens in this column. If there is more than one queen in column *i*, $\Sigma^{n}_{k=1} V_{ik}$ will be greater than 1, and the first term ($\Sigma^{n}_{k=1} V_{ik}$ - 1) will be positive. The effect is the opposite of no queen in the column. Furthermore, if there are many queens in the column, the magnitude of the effect will be even greater. The effect is to decrease V_{ij}, the number of queens in this column. The second term is exactly the same as the first term except that it is for row *j*.

The third and fourth terms inside of -{...} correspond to the upward and downward diagonals, respectively. The idea for these terms is somewhat similar to that of the first and second terms. For example, for the third term, $\Sigma_{k = \max [1-i,\, 1-j] \text{ to } \min [n-i,\, n-j] \text{ and } (k\neq 0)} V_{i+k,j+k}$, we add up the values of *V*'s along the upward diagonal except V_{ij}, the one in the column *i*, row *j* position. If there are one or more queens along the diagonal line, this term will be positive, i.e., -{$\Sigma_{k=..} V_{i+k,j+k}$} will be negative, and affects to reduce the value of u_{ij}, i.e., to make V_{ij} =0. If there is no queens along the diagonal line, this term will be zero, no effect to the value of u_{ij}, i.e., it means to keep the

current value of V_{ij}.

The energy function can be defined as:

$$E = \frac{1}{2}\left\{ \sum_{i=1}^{n}\left(\sum_{k=1}^{n} V_{ik} - 1 \right)^2 + \sum_{j=1}^{n}\left(\sum_{k=1}^{n} V_{kj} - 1 \right)^2 + \right.$$

$$\left. \sum_{i}\sum_{j}\sum_{k(k\neq 0)} V_{i+k,\, j+k} V_{ij} + \sum_{i}\sum_{j}\sum_{k(k\neq 0)} V_{i+k,\, j-k} V_{ij} \right\}$$

The sigmoid function for $V_{j\ell}$ can be defined as,

$$V_{j\ell} = \frac{1}{2}\left(1 + \tanh \frac{u_{j\ell}}{u_0} \right)$$

Since our goal is to make $V_{j\ell} = 0$ or 1, we can choose a small value of u_0 (e.g, 0.1) (Fig. 3.7).

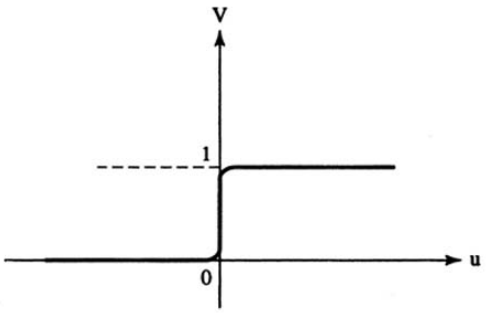


Fig. 3.7. The sigmoid function $V_{j\ell} = 1/2\ \{1 + \tanh\ (u_{j\ell} / u_0)\}$ with small u_0.

Alternatively, we can choose even a simpler function for $V_{j\ell}$ in terms of $u_{j\ell}$ as follows.

$$V_{j\ell} = \begin{cases} 1 & \text{for } u_{j\ell} \geq 0 \\ 0 & u_{j\ell} < 0 \end{cases}$$

Note. In this example, the equation of motion does not have $-u_{ij}$ and I_{ij} terms discussed for general cases.

Termination condition of iterations

We can continue until $E = 0$, rather than $E^{(n+1)} \geq E^{(n)}$ to avoid a possible local minima, where (n) represents the n-th iteration.

3.5.2 A General guideline to apply the Hopfield-Tank model to optimization problems

After studying a specific application of the Hopfield-Tank model to the n-queen problem, we are now in a better position to understand how to apply the model to other optimization problems. Here is a guideline.

- Define V_{ij} so that they can represent solutions of the problem. (e.g., $V_{ij} = 0$ means no queen, 1 means yes queen)

- Define an equation of motion as:

$$\frac{du_{ij}}{dt} = -\left\{\text{function of } V_{ij}\right\}$$

in such a way that u_{ij} tends to decrease when V_{ij} is supposed to decrease, and conversely, u_{ij} tends to increase when V_{ij} is supposed to increase.

- Values of w_{ij} are fixed by the equation of motion.

- Values of V_{ij} change through changes of u_{ij}. In turn, the changes of u_{ij} are affected by the changes of V_{ij}. The changes of V_{ij} are much slower than those of u_{ij}, because of commonly used activation function forms. For example, in Fig. 3.7, the value of V stays the same as either 0 or 1 for most values of u. That is, u_{ij} changes frequently; V_{ij} changes occasionally after significant changes of u_{ij}.
- Classical techniques typically use differential equations in terms of V_{ij} themselves, e.g., $dV_{ij}/dt = ...$, while the Hopfield-Tank uses u_{ij} as an intermediate stepping stone to solve V_{ij}. Biological systems use such "indirect control".

- The energy function can be derived based on the fundamental principle for the relationship between the equation of motion and the energy:

$$\frac{du_{ij}}{dt} = -\frac{\partial E}{\partial V_{ij}} \qquad \text{i.e.,} \qquad E = -\int \frac{du_{ij}}{dt} dV_{ij}$$

In many applications, it may be easier to write the equation of motion first from the given problem. Derive or conjecture the energy function in such a way that when it is differentiated, the result gives the equation of motion. If this is successful, it will be easier than directly integrating the equation of motion to obtain the energy function, since differentiation is typically easier than integration.

For example, in the above n-queen problem, differentiation of the first term of energy function leads to the corresponding first term of equation of motion as follows.

$$\frac{\partial}{\partial V_{ij}}\left[\frac{1}{2}\sum_{i=1}^{n}\left(\sum_{k=1}^{n}V_{ik}-1\right)^2\right]\rightarrow-\left(\sum_{k=1}^{n}V_{ik}-1\right)$$

For example, for $n = 2$

$$\left[\frac{1}{2}\sum_{i=1}^{n}\left(\sum_{k=1}^{n}V_{ik}-1\right)^2\right]=\frac{1}{2}\{(V_{11}+V_{12}-1)^2+(V_{21}+V_{22}-1)^2\}.$$

Then, for example,

$$-\frac{\partial}{\partial V_{21}}\left[\frac{1}{2}\sum_{i=1}^{n}\left(\sum_{k=1}^{n}V_{ik}-1\right)^2\right]=-\frac{1}{2}\left\{0+2\left(V_{21}+V_{22}-1\right)\cdot\frac{\partial V_{21}}{\partial V_{21}}\right\}$$

$$=-\left(V_{21}+V_{22}-1\right)=-\left(\sum_{k=1}^{2}V_{2k}-1\right)$$

3.5.3 Traveling Salesman Problem (TSP)

The traveling salesman problem (TSP) is an NP-complete problem, notorious for its time complexity. For this reason, the TSP has been chosen as a popular bench mark problem to test the effectiveness of many new techniques. Basically, earlier techniques are based on exhaustive search, which means "try all" to find an optimal solution. Although other popular techniques such as dynamic programming and branch-and-bound algorithms are better than pure exhaustive search, their principles are improvements over exhaustive search by cutting fruitless branches in the search space.

The Hopfield-Tank model is based on a totally new idea, very easy to implement on a parallel computer, and appears to find solutions very fast. This is why there was so much excitement when this model was announced and demonstrated to solve the TSP. The solutions are not guaranteed to be optimal, but they are said to be good enough for practical applications [Hopfield and Tank, 1985].

The basic idea of applying the Hopfield-Tank to the TSP is the same as we have seen in this section, such as the *n*-queen problem. We have to define V_{ij} appropriately so that it represents our solution. We also have to define an equation of motion so that it drives our solution in a desired direction. In the following, we will discuss the basic idea using a simple example.

Formulation of the Hopfield-Tank model for the TSP

Problem Given an undirected weighted graph, find a shortest tour by visiting every vertex exactly once.

We will illustrate the idea by using a specific example of $n = 4$ cities (Fig. 3.8). Of course, the method can be applied to any problem with any value of n.

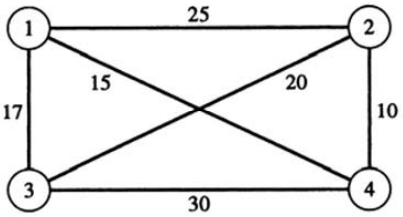


Fig. 3.8 A TSP example of $n = 4$ cities.

Define the "distance" matrix, $[d_{ij}]$, to represent the distance between each pair of the cities. The matrix is always symmetric, so the upper triangular matrix is sufficient. When two cities are not directly connected, an arbitrarily large distance can be assigned so that any solution connecting unconnected cities will disappear soon because of the penalty. For our specific example, the distance matrix will be given as follows:

Distance matrix

City / City	1	2	3	4
1	0	25	17	15
2		0	20	10
3			0	30
4				0

One optimal solution is given as:

17	15	10	20		Total
3 ———	1 ———	4 ———	2 ———	(3)	62

We note that there is always another corresponding optimal solution for each optimal solution by traversing the route in the opposite direction (in the above, 3 — 2 — 4 — 1 — (3)).

Our problem is to determine an order of cities to be visited that gives the minimum traveling distance. To represent a solution (not necessarily optimal) which shows the order of cities to be visited, we consider an $n \times n$ "solution" matrix. For our example,

if a solution is to visit Cities No. 3, 1, 4, 2 (and 3) in this order, then the 4×4 matrix will be:

A solution matrix

City \ Position	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	0	0	0
4	0	0	1	0

Here entry 1 means visit; 0 means not visit. More precisely, an entry 1 in the *i*-th row, *j*-th column means City *i* is the *j*-th city to be visited. For example, City 3 will be visited first, City 1 will be visited second, and so on. Note that there is exactly one 1 in each row and column. We define V_{Xi}, $0 \le V_{Xi} \le 1$, to represents the *degree* of visiting City *X* at Position *i* (i.e., as *i*-th city). If $V_{Xi} = 0$ then not visit City *X* as the *i*-th city; if $V_{Xi} = 1$ then visit; if V_{Xi} is between 0 and 1, say 0.3, then visit with degree 0.3 (although in real life, you cannot do this). For example, during iteration processes, our solution matrix can look as follows:

City *X* \ Position *i*	1	2	3	4
1	0.1	0.8	0	0.1
2	0.2	0	0.2	0.8
3	0.7	0.1	0	0.2
4	0.3	0.4	0.9	0.2

The basic equations can be defined as:

$$\frac{du_{Xi}}{dt} = -\frac{u_{Xi}}{\tau} - A\sum_{j \neq i} V_{Xj} - B\sum_{Y \neq X} V_{Yi} - C\left(\sum_{Y}\sum_{j} V_{Yj} - n\right)$$
$$-D\sum_{Y} d_{XY}\left(V_{Y,\,i+1} + V_{Y,\,i-1}\right).$$

$$V_{Xi} = g(u_{Xi}) = \frac{1}{2}\left\{1 + \tanh\frac{u_{Xi}}{u_0}\right\}.$$

$$E = \frac{1}{2}A\sum_X\sum_i\sum_{j\neq i} V_{Xi}V_{Xj} + \frac{1}{2}B\sum_i\sum_X\sum_{X\neq y} V_{Xi}V_{Yi}$$

$$+\frac{1}{2}C\left(\sum_X\sum_i V_{Xi} - n\right)2 + \frac{1}{2}D\sum_X\sum_{Y\neq X}\sum_i d_{XY}V_{Xi}\left(V_{Y,\,i+1} + V_{Y,\,i-1}\right)$$

The coefficients A, B, C, and D, and u_0 are constants (for example $A = B = D = 500$, and $C = 200$). We have five terms on the right-hand side of the first equation of motion. The first term is called biological term. The value of τ can be set to 1 without loss of generality (see [Hopfield and Tank 1985]. The second term drives to have only one 1 in Row X (if other elements are 0, then keep the current value of u_{Xi}; otherwise, reduce u_{Xi} so that V_{Xi} becomes 0). The third term is the same as the second except that it is for Column i. The fourth term is for having exactly n 1's in the entire solution matrix.

The last term is to minimize the total distance to be traveled by the solution. If V_{Xi} and $V_{Y,\,i+1}$ are close to 1, the degree of visiting these two cities in sequence (ith visit for X and (i+1)st for Y) is high. If d_{XY}, the connecting distance between cities X and Y, is also large, it is better to get rid of V_{Xi} by reducing u_{Xi}. Similarly, $V_{Y,\,i-1}$ is for (i-1)st visit for Y and ith visit for X. In our example, suppose that $X = 1$ and $i = 2$; then

$$\sum_Y d_{1Y}\left(V_{Y3} + V_{Y1}\right) = d_{12}\left(V_{23} + V_{21}\right) + d_{13}\left(V_{33} + V_{31}\right) + d_{14}\left(V_{43} + V_{41}\right).$$

For example, if $V_{Xi} = V_{12}$ and $V_{Y,\,i+1} = V_{23}$, and if both are close to 1, then the degree of visiting these two cities in sequence is high: $X = 1$ for the 2nd city and $Y = 2$ for the 3rd city. Since $d_{XY} = d_{12} = 25$ is large, we would try to remove $V_{Xi} = V_{12}$.

3.6 The Kohonen Model

The Kohonen neural network model possesses interesting characteristics such as self-organization and competitive learning. The major objective in this section is to study these characteristics in the Kohonen network.

Background

One interesting aspect in the study of neural networks is how a neural network learns. The backpropagation model, discussed in the previous chapter, performs *supervised learning*. For each input pattern, the neural network computes its output. At the same time, the neural network is given correct output by the teacher, compares it with its own output, and tries to learn how to make its output closer to the correct output. This

supervised learning process is like a private lesson with a tutor. Each time the student comes up with an output (for example, pronouncing a word or carrying out an arithmetic computation), the tutor immediately gives the correct answer.

A variation of this supervised learning is called **graded** or **reinforcement learning**. In this method, the neural network is not given correct output corresponding to each input. Instead, the neural network is occasionally given a "grade" or "score" for overall performance for its outputs since the last time it was graded. Graded learning is analogous to a classroom situation, where students are occasionally given quizzes and exams, and the resulting scores reflect their overall performance.

When we think of the way humans learn, the above-mentioned supervised and graded learning are certainly important. But human learning is not limited only to these forms and involves much more. For example, a baby gains a tremendous amount of knowledge early on, such as how Mom, Dad and other objects around the baby look, sound, smell and feel. Obviously the baby does not learn this by being told what is correct and what is not. In other words, humans have the ability to learn without being supervised or graded. It will be interesting to model such human learning in neural networks, as, for example, in the form of **unsupervised learning** (that is, neither supervised nor graded). The term unsupervised learning refers to a method in which the neural network can learn by itself without external information during its learning process.

Self-organization is an unsupervised learning method, where the neural network *organizes itself* to form useful information. This method may model some of the human learning processes discussed above, which are neither supervised nor graded.

In **competitive learning**, neurons (or connecting edges) compete with each other. The winners of the competition strengthen their weights while the losers' weights are unchanged or weakened. The idea is somewhat similar to the principle of evolution, which will be discussed in the next chapter on genetic algorithms; the winning species in the evolution process survives while the losers become extinct. By employing competitive learning, the neural network can self-organize, achieving unsupervised learning. These terms "self-organization" and "competitive" learning will become clearer when we study the Kohonen network in the following.

Architecture

The architecture of the Kohonen network is shown in Fig. 3.9. It is a multilayered network of two layers; the first is the input layer and second is the output layer, called the **Kohonen layer**. The neurons in the Kohonen layer are called **Kohonen neurons**. Every input layer neuron is connected to every Kohonen neuron, with a variable associated weight. The network is non-recurrent, that is, feedforward (input information propagates only from the left to right). Continuous (rather than binary or bipolar) input values representing patterns are presented sequentially in time through the input layer, without specifying the desired output. Each pattern is represented in the form of a vector, $\mathbf{x} = (x_1, x_2, ..., x_n)$.

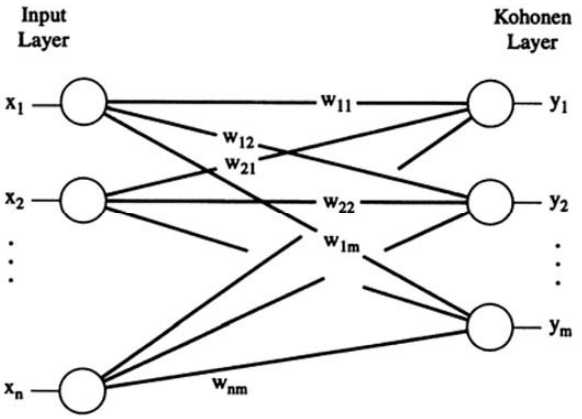


Fig. 3.9. The Kohonen network architecture.

In the above figure, the neurons in the Kohonen layer are arranged in one dimension. They can also be arranged two-dimensionally. In both one and two-dimensional Kohonen layer configurations, a "neighborhood parameter" or "**radius**," r can be defined to indicate the neighborhood of a specific neuron. Fig. 3.10 illustrates examples of defining radiuses for one and two-dimensional Kohonen layer configurations. Neighbors do not wrap around cyclically from one end to the other, i.e., missing end neurons, if any, are not considered. For two-dimensional configurations, other shapes such as hexagons can also be defined. There are variations for the Kohonen network other than the model described here in the architecture and computational procedure.

Computational procedures

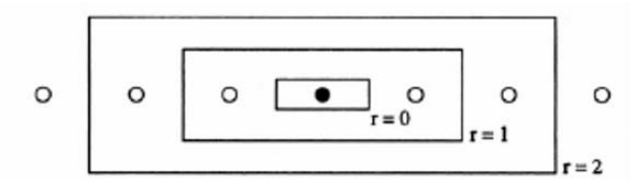
Initialization (Step 0).

Assign small real random values to weights, w_{ij} for $i = 1$ to n and $j = 1$ to m.
Initialize the following two parameters:
A neighborhood parameter, radius r (e.g., $r = 3$). (See Fig. 3.10.)
A learning rate, α, where $0 \leq \alpha \leq 1$ (e.g., $\alpha = 0.8$).

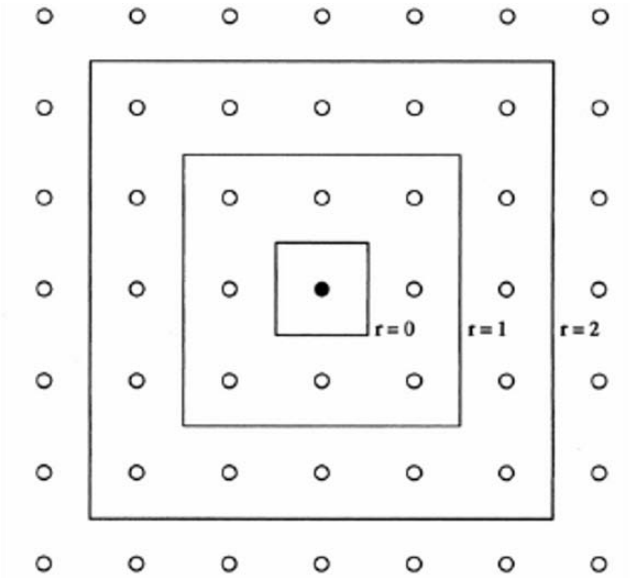
Iterations

Repeat the following Steps 1 through 5 for a sequence of input vectors, **x**'s, drawn at random.

Step 1. Enter a new input vector, $\mathbf{x} = (x_1, x_2, ..., x_n)$, to the input layer.



(a)



(b)

Fig. 3.10.Neighborhood examples of neuron ● in Kohonen layers represented in terms of radiuses. (a) A one-dimensional configuration. (b) A two-dimensional configuration.

The following Steps 2 and 3 perform *competitive learning*.

Step 2. Selection of a winning Kohonen neuron.

The Kohonen neurons compete on the basis of which of them have their associated weight vectors, $\mathbf{w}_j = (w_{1j}, w_{2j}, ..., w_{nj})$, "closest" to $\mathbf{x}$, as measured by a "distance function," $D(\mathbf{w}_j, \mathbf{x})$. Each Kohonen neuron, j for $j = 1$ to m, calculates its distance as $D(\mathbf{w}_j, \mathbf{x})$. There are different choices for the function form of $D(\mathbf{w}_j, \mathbf{x})$. A common form is:

$$D(\mathbf{w}_j, \mathbf{x}) = \sum_{i=1}^{n} \left(w_{ij} - x_i \right)^2 \quad .$$

The winning Kohonen neuron is the one with the smallest distance.

Step 3. Weight modification.

For all neurons, j, within a specified neighborhood radius of the winning neuron, adjust the weights according to the following formula:

$$\mathbf{w}_j^{(t+1)} = \mathbf{w}_j^{(t)} + \alpha(\mathbf{x}^{(t)} - \mathbf{w}_j^{(t)})$$

This weight modification moves the weights associated with the winning neurons a fraction of α of the way from $\mathbf{w}_j$ to $\mathbf{x}$. For example, in extreme cases, if $\alpha = 1$ then $\mathbf{w}_j^{(t+1)}$ will change to $\mathbf{x}^{(t)}$; if $\alpha = 0$ then $\mathbf{w}_j^{(t+1)}$ will remain as $\mathbf{w}_j^{(t)}$. For a typical value of α, which is between 0 and 1, $\mathbf{w}_j^{(t+1)}$ will be between $\mathbf{x}^{(t)}$ and $\mathbf{w}_j^{(t)}$.
For all the remaining (losing) neurons, the weights are unchanged, i.e., $\mathbf{w}_j^{(t+1)} = \mathbf{w}_j^{(t)}$.

Step 4. Update learning rate α. Typically, α is gradually reduced over iterations.

Step 5. Slowly reduce radius r at specified iterations.

Example.

We consider a simple scenario of a one-dimensional Kohonen neural network with three input layer neurons x_1, x_2, x_3 and 10 Kohonen layer neurons y_1, y_2, ..., y_{10}. We interpret each input vector $\mathbf{x} = (x_1, x_2, x_3)$ as representing a color: (1, 0, 0) = red, (0, 1, 0) = yellow, (0, 0, 1) = blue, (1, 1, 0) = orange, etc. We may have a sequence of, say, 20 input vectors.

Input vector No.17 may randomly be picked up first, representing red, $\mathbf{x} = (x_1, x_2, x_3) = (1, 0, 0)$. We will determine the winning Kohonen neuron, i.e., y_j that "best represents" this $\mathbf{x}$. To do so, we compute 10 associated distances corresponding to the 10 Kohonen neurons and find y_j that gives the minimum distance. Perhaps y_6 is the winner; the associated weight vector $(w_{16}, w_{26}, w_{36}) = (0.9, 0.1, 0.2)$ is closest to this input vector, $\mathbf{x} = (x_1, x_2, x_3) = (1, 0, 0)$, and its associated distance is:

$$\text{Distance} = \sum_{i=1}^{3} (w_{i6} - x_i)^2 = (0.9 - 1)^2 + (0.1 - 0)^2 + (0.2 - 0)^2 = 0.06.$$

Next, we adjust the weights for neighboring ys; e.g., y_5, y_6, y_7 if $r = 1$. The new weights will be between the current weights and $\mathbf{x}$. Assuming α is 0.8, the new weights associated with y_6 will be:

$$w_{16}^{(\text{new})} = w_{16} + \alpha(x_1 - w_{16}) = 0.9 + 0.8 * (1 - 0.9) = 0.9 + 0.08 = 0.98.$$
$$w_{26}^{(\text{new})} = w_{26} + \alpha(x_2 - w_{26}) = 0.1 + 0.8 * (0 - 0.1) = 0.1 - 0.08 = 0.02.$$
$$w_{36}^{(\text{new})} = ...$$

We notice, for example, that the new weight $w_{16} = 0.98$ is between the current weight $w_{16} = 0.9$ and $x_1 = 1$. The new weights associated with y_5 and y_7 will also be computed similarly.

The second input vector may be No. 12 and represent blue, $\mathbf{x} = (0, 0, 1)$. Perhaps y_3 is the winner. The third input vector may represent red, $\mathbf{x} = (1, 0, 0)$. …… After 20 input vectors (an epoch), reduce α from 0.8 to e.g., 0.8 * 0.9 = 0.72. After more epochs, the weight vectors converge, i.e., the neural network self-learns by clustering the three colors. If input is red, y_6 fires; if input is blue, y_3 fires; and so on.

What does the Kohonen network accomplish?

The network stores the presented input vector patterns through modification of weight vectors. After enough input vectors are presented, the weight vectors become densest where input patterns are most common, and become least dense where input patterns are rare. That is, the effect is to **cluster** (or categorize) the input patterns. The density distribution of the weight vectors tends to approximate the density distribution of the input vectors. In addition, similar input patterns will be classified in the same cluster and will fire the same output neurons. The input patterns are stored and classes are found by the network itself without a teacher or ideal output patterns; this is the idea of self-organization and unsupervised learning.

Application examples

A neural phonetic typewriter

Kohonen [1988] shows a speaker-adaptive speech recognizer using a Kohonen neural network. Spoken words are presented to the neural network through a microphone with some pre-processing. The output neurons are labeled with phonemes. Basically, the network trains itself to map the input to output.

Data compression as a vector quantizer

A Kohonen network can be used to compress and quantize data, such as for speech and images, before storage or transmission to reduce the amount of information to be stored or sent. The principle is to categorize a given set of input patterns into classes, and represent any pattern by the class into which it fits. Generally, a neural network learning process of this type, to divide a set of input patterns into disjoint classes, is called **learning vector quantization**.

3.7 Simulated Annealing

Simulated annealing is a general technique for optimization problems in many application domains. Our interest in this chapter is its application to neural networks, particularly to Hopfield networks, but simulated annealing can be employed in many other areas. Loosely speaking, Boltzmann machines, which will be discussed in the next section, are extensions of Hopfield networks in which the simulated annealing technique is incorporated to achieve optimization.

What is simulated annealing and why do we use it?

When we solve a hard optimization problem by an iterative process, the solution may prematurely stuck in an undesirable local minimum before reaching the global minimum (or an acceptable local minimum), as illustrated in Fig. 2.10. Simulated annealing is a technique to avoid undesirable local minima. In general, many iterative optimization techniques may determine a solution that minimizes or maximizes a certain objective function of a system, such as total error, energy, cost, or profit. Minimization and maximization are technically identical, i.e., exactly the same technique can be used – to minimize we go down and to maximize we go up.

The basic idea of employing iterative techniques is as follows. If we can find a solution that minimizes the objective function in one step, that would be the best. In order to find an x that minimizes the function $f(x)$ in calculus, we set the derivative $f'(x) = 0$. This is a one-step procedure. For many difficult problems, however, this one-step approach does not work; in such cases, we employ an iterative procedure, such as that discussed earlier for the backpropagation model and Hopfield networks. Typically, we start with an arbitrary, often randomly selected, solution and improve the solution incrementally through many iterations. In each step $f(x)$ decreases slightly, and hopefully $f(x)$ reaches the absolute bottom. A common problem of this technique is that the solution may prematurely terminate upon reaching a local basin that is located at a high elevation, and we may mistakenly conclude that this is the best solution.

Simulated annealing tries to overcome this local minima problem by incorporating probabilistic, rather than strictly deterministic, approaches in search of optimal solutions. The term “probabilistic” is also called “statistical” or “stochastic,” all of which describe the same concept in this context. The name "simulated annealing" is used since it is analogous to a gradual cooling or annealing process of a metal or another substance to its lowest energy level, resulting in a crystal. If the metal is heated to a high temperature, it melts. If the metal is cooled quickly (quenching), atoms or molecules bind together without reaching the lowest binding energy levels, leading to an amorphous or defective state. This is analogous to an iterative process that is trapped in an undesirable local minimum. We want to produce a crystal state, a global minimum, by annealing. Therefore, we will set the system temperature high at the beginning, gradually reducing the temperature, making certain that the system is near thermal equilibrium at each temperature.

The algorithm discussed here is analogous to physical phenomena, especially statistical mechanics. Energy E is a measure that is used to determine whether a minimum (or maximum) solution has been reached. E can represent total error, cost, profit, etc., depending on the specific application. Typically, E does not represent physical energy. Similarly, temperature T is another measure often called pseudo-temperature, a parameter to perform simulated annealing computer algorithms. That is, T is analogous to temperature in thermodynamics, but typically it is not physical.

Probabilities of states in terms of the energy and the temperature

We briefly overview the statistical mechanics aspect from which simulated annealing is conceived. In thermodynamics, the probability of finding the system in a particular state with energy E and temperature T is proportional to the Boltzmann probability factor:

$$e^{-\frac{E}{KT}}$$

where k is the Boltzmann constant, 1.3896 × 10^{-23} joule/kelvin, and T is a measurement in kelvin = ^oC + 273.15. Consider two states S_1 and S_2, with energy E_1 and E_2, and the same temperature T. The ratio of the probabilities of the two states is as follows:

$$P(S_1)P(S_2) = \exp\frac{\left(-\frac{E_1}{kT}\right)}{\exp\left(-\frac{E_2}{kT}\right)} = \exp\frac{\left[E_1 - E_2\right]}{kT}.$$

For example, a molecule of gas in the earth's atmosphere has its lowest energy at the sea level of 0 meters and higher energy at a higher altitude. But the probability of finding the molecule 10 meters above sea level is about the same as at the sea level, because the energy difference [E_1 - E_2] is very small in comparison with kT, i.e., the probability ratio of these two levels is exp(-[E_1 - E_2]/kT) ≈ exp(- 0/kT) ≈ 1. But when the altitude becomes much higher, say, 10,000 meters, the probability difference is significant. At such a high altitude, significantly fewer molecules exist. The probability of a molecule, or, equivalently, the number of molecules, decreases exponentially with the altitude.

Now, as a hypothetical scenario, assume that the temperature is increased 1,000 times. That is, T in exp(-[E_1 - E_2]/kT) is 1,000 times larger. Under such circumstance, the probability of finding the molecule would be much higher, even at a high altitude. The probability at 10,000 meters is the same as at 10 meters at the original temperature. Accordingly, when temperature is high, the system explores a large number of possible states, ranging at low to high altitudes.

Simulated annealing adopts this thermodynamics concept. We do not need to keep the Boltzmann constant, k, or to measure E in joules and temperature in kelvin, since we are not dealing with physical systems. We can drop k by selecting an appropriate ratio between E and T. We start with a high temperature so that a large number of possible solutions can be explored at an early stage. We lower the temperature gradually, as during metal annealing, ensuring a low-energy solution at each temperature.

Simulated annealing algorithm

Algorithm

Step 0. Initialization.
Randomly select a solution vector **x**. Set the temperature parameter T to T_0. We may select T_0 large enough in comparison with a representative | ΔE | so that $e^{-\Delta E/T}$ is sufficiently close to 1. ΔE is defined in the next Step.

Step 1. Beginning of the outer and inner loops.
Compute xp, a perturbed (slightly changed) solution of x.
Determine $\Delta E = E(\mathbf{x}_p) - E(\mathbf{x})$, the change in the energy (objective) function.

Step 2. Select the current **x** (i.e., no change) or $\mathbf{x}_p$ (i.e., the perturbed new solution) based on the following criteria for a new **x** of the next time step.
Case 1. $\Delta E < 0$, i.e., $\mathbf{x}_p$ is better than **x**. Select $\mathbf{x}_p$.
Case 2. $\Delta E \geq 0$, i.e., $\mathbf{x}_p$ is not better than **x**. Select $\mathbf{x}_p$ with the probability of $e^{-\Delta E/T}$, and **x** with the probability of $1 - e^{-\Delta E/T}$. We can implement Case 2 by picking up a random number r on [0, 1], then selecting $\mathbf{x}_p$ if $r < e^{-\Delta E/T}$, and selecting **x** otherwise.

Step 3. Repeat Steps 1 and 2 until $|\Delta E|$ becomes small enough, i.e., the system is near equilibrium at this temperature. (Alternatively, repeat until the number of iterations exceeds a predetermined maximum number).

Step 4. Reduce the temperature T and repeat Steps 1 through 3 until T reaches zero or a small positive number. Possible schemes for reducing the temperature will be discussed later.

In the above algorithm, Step 2, Case 2 is a key to simulated annealing. This action forces the relative probabilities of the two states of xp and x, that differ in energy by (E, to match the Boltzmann distribution [Kirkpatrick, 1988]. The idea of the process is that even if (E (0, i.e., the new solution is the same or worse than the current one, we still select the new solution with a certain probability. This helps to escape from local minima by not focusing solely on downward movement. The probability of this selection is high when temperature T is large, since $e^{-\Delta E/T} \approx e^{-\Delta E/\infty} \approx e^0 \approx 1$. The probability becomes smaller when T gradually decreases, and for $T \rightarrow 0$, the probability $e^{-\Delta E/T} = 1/e^{\Delta E/T} \approx 1/e^{\infty} \rightarrow 0$. (When both T and $|\Delta E|$ are exactly zero, $e^{-\Delta E/T}$ is undefined. We can avoid this extreme case by stopping the algorithm when (E and T are sufficiently small.) The effect of this probability change is that, starting at a high temperature, we try to jump out of local minima more aggressively during an early stage. Later, presumably most, if not all, local minima have been escaped, and when the system is close to a global minimum, we perform a more gentle minimizing process.

Example. Traveling salesman problem (TSP)

N = 5 cities are randomly scattered within a 1.0 (1.0 square area, and numbered 1 through 5 (Fig. 3.11.) Each solution vector x is a permutation of N numbers. E(x) is the total distance for solution x.

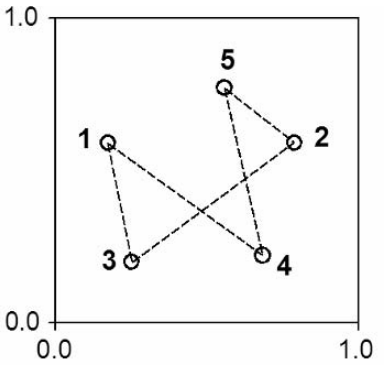


Figure 3.11. A randomly selected solution example of a traveling salesman problem of five cities. The solution vector $\mathbf{x}$ = (1, 3, 2, 5, 4).

Application of the algorithm

Step 0 Randomly select a solution vector $\mathbf{x}$, such as (1, 3, 2, 5, 4), for the 0th iteration. Set T to T_0.

Step 1. Compute $\mathbf{x}_p$, a perturbed solution of $\mathbf{x}$. For example, $\mathbf{x}_p$ may be obtained by randomly swapping two cities in $\mathbf{x}$. Perhaps it is (1, 3, 4, 5, 2) for the first iteration. Determine $\Delta E = E(\mathbf{x}_p) - E(\mathbf{x})$, the change in the total distance.

Step 2. Case 1. If $\Delta E<0$, i.e., $\mathbf{x}_p$ is a better solution than $\mathbf{x}$, select $\mathbf{x}_p$ as a new $\mathbf{x}$ for the next step.
Case 2. IF $\Delta E \geq 0$, select $\mathbf{x}_p$ with $e^{-\Delta E/T}$ probability, keep current $\mathbf{x}$ with $1-e^{-\Delta E/T}$ probability.

Step 3. Repeat Steps 1 and 2 until $|\Delta E|$ is small enough.

Step 4. Reduce T by, for example. (new T) = 0.9 × (current T). Repeat Steps 1 though 3. Terminate the entire algorithm when T reaches zero or a small number.

My graduate student Bob Crichton performed numerical experiments on TSP, by selecting various values of parameters such as the number of cities (5, 10, 15, 20), initial temperature T_0 (e.g., 0.02, 0.05), $|\Delta E|_{\min}$ in Step 3 (e.g., 0.1, 0.21, 0.42), and the coefficient of the temperature reduction formula in Step 4 (e.g., 0.9, 0.95, 0.99). He also experimented with a slightly different way of obtaining a perturbed solution in Step 1 – randomly picking two cities then reversing the order of the cities between them including the two cities. For example, if Cities 3 and 5 are picked in 6-city solution (1, 3, 4, 6, 5, 2), then the perturbed solution will be (1, 5, 6, 4, 3, 2). This approach often yields more moderate changes in the total distance than the one described in Step 1. This is because the former involves changing two distances (in the above example, the distances between 1 and 3, and 5 and 2), where as Step 1 involves changing four distances.

Fig. 3.12 shows a typical run to obtain an optimal solution for 10 cities with T_0 = 0.021, $|\Delta E|_{\min}$ = .021, and (new T) = 0.9 × (current T), employing Step 1 algorithm for a perturbed solution. This run required a few seconds of runtime on a 2 Ghz PC.

It is interesting to observe the energy changes in Fig. 3.12 (a) where local minima were escaped due to the simulated annealing effect. When we select too small a coefficient value in Step 4 (e.g., 0.8), temperature decreases rapidly, and consequently the simulated annealing effect declines quickly and we may not be able to reach an optimal solution.

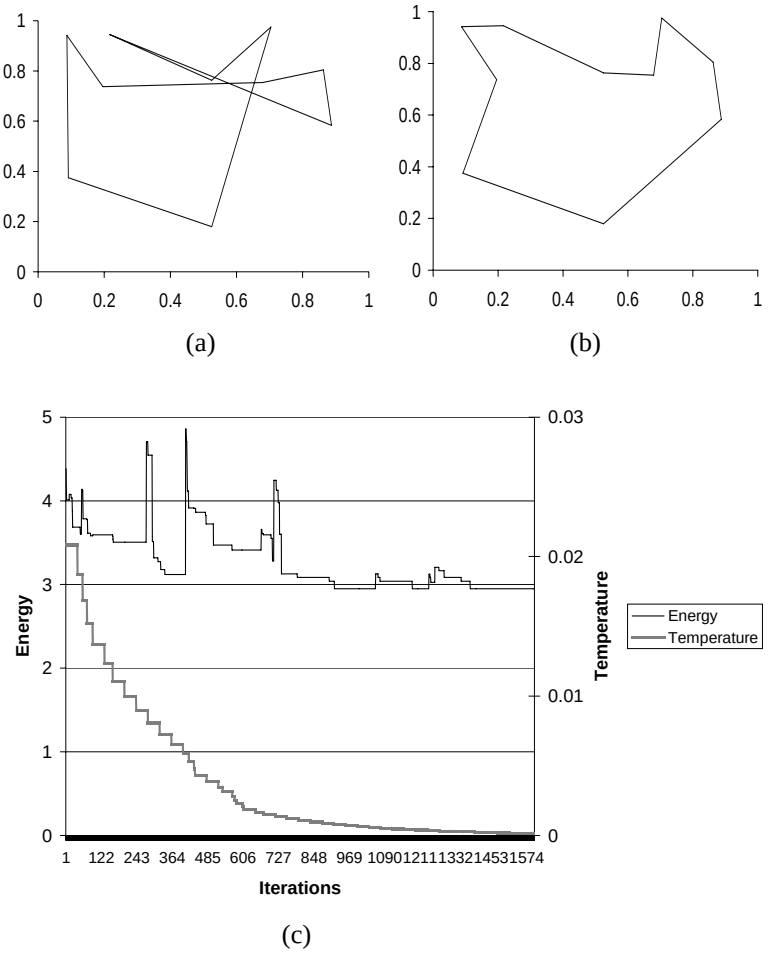


Fig. 3.12. Simulated annealing experiment on TSP with 10 cities. (a) Initial solution. (b) Optimal solution. (c) Total energy (distance) and temperature changes over iterations.

Considerations on reducing the temperature

In Step 4 of the algorithm, we reduce the temperature T, that is, we perform annealing (cooling). Various schemes have been proposed for this process. There is a tradeoff between the cooling speed and finding the optimal solution. That is, if the cooling speed is very slow, the global minimum may be guaranteed, but it may require much,

often impractically long, computation time. The following is such a very slow cooling scheme [Geman and Geman, 1984]:

$$T_t = T_0/\log(1 + t) \qquad t = 1, 2, \ldots$$

where t represents the t^{th} iterate of the outer loop.
On the other hand, if the cooling speed is faster, the computation time will be shorter, but finding the global minimum may not be guaranteed. Even so, often near-optimal solutions are practically good enough. The following is such a scheme used in Step 4 of Example (TSP) above [Kirkpatrick, 1988]:

$$T_t = \alpha T_{t-1}$$

where the reducing factor α, $0 < \alpha < 1$, is typically $0.8 \leq \alpha \leq 0.99$.

3.8 Boltzmann Machines

3.8.1 An Overview

A **Boltzmann machine** can be defined as an extension of the Hopfield network. One major distinction between these two models is the ways the states of the neurons are updated. In the Hopfield network, the neurons are updated according to a *deterministic* formula ($x_i(t + 1) = 1$ if $net_i(t) > \theta_i$, …, given in Section 3.3). Note that the term deterministic refers to the equations to change the *states* of neurons, rather than how the neurons are selected for the update. Similarly, u_i for the Hopfield-Tank model is evaluated in a deterministic manner. In the Boltzmann machine, the states of the neurons are updated in a *stochastic* way using simulated annealing. The probabilities in simulated annealing are given by the Boltzmann distribution in statistical mechanics; hence the name Boltzmann machine. Additional major distinctions between the Boltzmann machine and the Hopfield network are: the types of learning and the existence of hidden neurons [Ackley, et al, 1985].

Summarizing the three major differences between the Boltzmann machines and the Hopfield network are (more explanations below):

1. Neuron update. The Boltzmann machines - stochastic using simulated annealing, while the Hopfield network - deterministic.

2. Learning forms. The Boltzmann machines can be either supervised or unsupervised. The Hopfield has only one learning form.

3. Hidden neurons. The Boltzmann machines have hidden neurons, but the Hopfield network does not.

The Boltzmann machines can be classified into two types based on the ways the networks learn – **supervised** and **unsupervised** (sometimes called **self-supervised**). The supervised version is similar to the backpropagation model, but requires much more computation time. For this reason the Boltzmann machine for supervised

learning is not extensively employed in practice and the details will not be discussed in this book.

Another aspect of Boltzmann machines is how the neurons are grouped. In the unsupervised model, the neurons are divided into two groups – **visible** and **hidden**. In the supervised model, the neurons are also divided into two groups, visible and hidden; furthermore, the visible neurons are subdivided into two subgroups – **visible input** and **visible output**. In addition to the above two aspects, namely the ways the networks learn and how the neurons are grouped, there are other features that characterize Boltzmann machine models. They include: how neurons are connected among and within the groups – fully or partial; and whether each connection is bidirectional (i.e., recurrent) or unidirectional.

The subject of the Boltzmann machines is more complicated than other neural network models we have seen. There are two major reasons for this situation. First, as discussed above there are many choices for the network characteristics (e.g., the network architecture and the ways the networks learn) leading to many Boltzmann machine models. Different models are useful for various applications. Second, the learning process of a Boltzmann machine requires more steps that involve various aspects of computation. In the following we will discuss only the most widely used models, in particular the unsupervised model.

The common computational characteristics of Boltzmann machines and the Hopfield network are as follows.

1. the weights are symmetric, i.e.,$w_{ij} = w_{ji}$;
2. no self-feedback, i.e., $w_{ii} = 0$;
3. the state of each neuron is binary (0 or 1) or bipolar (-1 or 1), representing ON or OFF, respectively;
4. the neurons are picked out randomly one at a time for update.

General problem description

Briefly stated, we are given a set of input patterns (vectors), collectively called the environment (hereafter, we use the terms “patterns” and “vectors” interchangeably). These input patterns are clamped (i.e., placed in effect forcefully) one at a time to the visible neurons in the case of the unsupervised learning model and visible input neurons in the supervised model. The objective is to match the probability distribution of the environment (i.e., input patterns) and the probability distribution of the network. We achieve this task by utilizing the hidden neurons and by training the network through adjustment of weights. When it is done, the probability distributions of the environment and the network will be the same. In effect, the input patterns will be clustered in the network according to a Boltzmann distribution.

In the following we discuss the architecture, problem description, learning process basics, and learning algorithm.

3.8.2 Unsupervised Learning by the Boltzmann Machine: The Basics

Architecture

Fig. 3.13 depicts a typical Boltzmann machine for unsupervised learning. The

neurons are classified into two groups – visible and hidden. Each neuron assumes one of two values, +1 or -1. The neurons are fully connected, i.e., every neuron is connected to every neuron in both groups. Every connection is bidirectional, i.e., the network is recurrent. The visible neurons are clamped to external states when the network interacts externally with its environment. The hidden neurons operate internally to make the network learn the underlying constraints imposed by the external inputs.

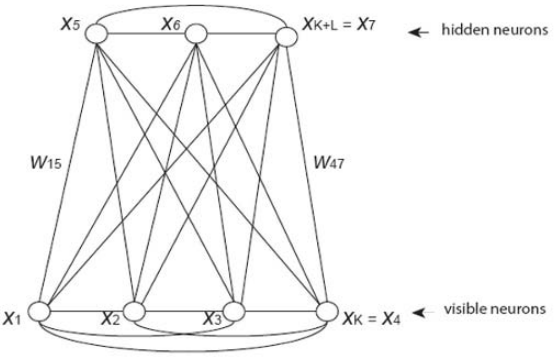


Figure 3.13. A Boltzmann machine architecture example of unsupervised learning.

There are K visible neurons at the bottom layer, whose state can be represented by $(x_1, \ldots, x_K)$, and L hidden neurons at the top layer, whose state can be represented by $(x_{K+1}, \ldots, x_{K+L})$. Each x_i assumes +1 or -1. The neurons are fully connected and each edge is bidirectional (recurrent).

Let the number of the visible neurons be K and the hidden neurons be L. Then there are a total of $K+L$ neurons in the network (in Fig. 3.13, $K = 4$, $L = 3$, and $K+L=7$). The state of these neurons can be represented by the state vector $\mathbf{x} = (x_1, \ldots, x_K; x_{K+1}, ..., x_{K+L})$. We can divide $\mathbf{x}$ into two groups, visible and hidden, as $\mathbf{x}_\alpha = (x_1, \ldots, x_K)$ and $\mathbf{x}_\beta = (x_{K+1}, \ldots, x_{K+L})$, respectively. To explicitly indicate vector $\mathbf{x}$ contains both visible and hidden neurons, we also write $\mathbf{x}$ as $\mathbf{x}_{\alpha\beta}$. A vector $\mathbf{x}$ represents a snapshot for a state of the network. For example, in Fig. 3.13, perhaps $\mathbf{x} = (1, -1, -1, 1; 1, -1, 1)$.

Problem description

In words, the basic idea of the problem is as follows. The problem involves two major components: a Boltzmann machine neural network and its environment (Fig. 3.14). The network has a type of architecture as previously depicted in Fig. 3.13, i.e., it consists of a set of visible neurons and a set of hidden neurons. The environment is a set of input patterns. Only the visible neurons are directly affected by the environment. The hidden neurons are affected by the environment only indirectly through the visible neurons.

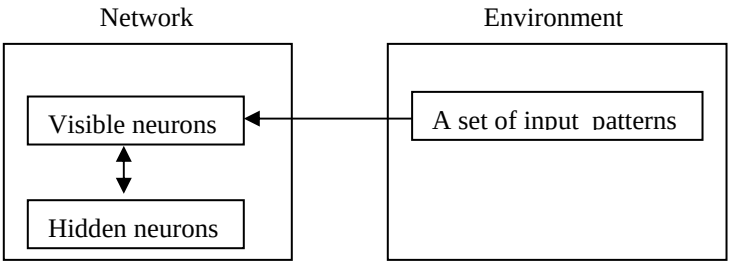


Fig. 3.14. The problem domain consists of a network and its environment.

The objective of the problem, in abstract terms, is to let the network create a model of the structure implicit in the set of input vectors by using the hidden neurons. More specifically, the implicit structure is the probability distribution of the set of input vectors. We want this probability distribution to be closely (or exactly) realized by the visible neurons when the network is running free from the environment. For example, if we have a set of three input vectors, (1, -1, -1, -1), (1, -1, -1, -1), (-1, -1, -1, 1) in the environment, the probability distribution will be 2/3 for (1, -1, -1, -1) and 1/3 for (-1, -1, -1, 1). When the model created by the network is perfect, the visible neurons will have exactly the same probability distribution.

More generally, we are given a set of Q training input patterns: $S = \{\mathbf{x}_{v1}, ..., \mathbf{x}_{vQ}\}$, where these patterns are not necessarily distinct. Patterns can be repeated in proportion to how often they are known to occur. Since each pattern has K components (x_1 through x_K), the entire set S of Q training patterns consists of QK components of 1s and -1s. Our task is to perform unsupervised learning on the network so that the visible neurons and the input patterns are *clustered* in terms of their probability distributions. Clustering means dividing the patterns into groups, where similar patterns are placed in the same group while all the others are in different groups. Clustering is performed by adjusting network weights w_{ij} as we will see.

Simple example

$K = 4$, $L = 2$, i.e., the architecture is obtained by removing x_7 in Fig. 3.13. A state of the 6 neurons, where each neuron takes +1 or -1, can be represented by the state vector $\mathbf{x} = (x_1, \ldots, x_4; x_5, x_6)$. We can divide $\mathbf{x}$ into two groups, $\mathbf{x}_\alpha = (x_1, \ldots, x_4)$ and $\mathbf{x}_\beta = (x_5, x_6)$. We assume $Q = 3$ training input patterns: (1,-1,-1,-1), (1,-1,-1,-1), (-1,-1,-1,1). The first two patterns are repeated. Set S of these input patterns is given by $S = \{\mathbf{x}_{v1}, \mathbf{x}_{v2}, \mathbf{x}_{v3}\} = \{(1,-1,-1,-1), (1,-1,-1, -1), (-1,-1,-1,1)\}$, which consists of $QK = 3 \times 4 = 12$ values of 1 or -1.

The visible neurons can be interpreted as representing colors: x_1 = red, x_2 = yellow, x_3 = green and x_4 = blue. (1,1,-1,-1) would represent orange. The above three input patterns represent red, red, blue. When these three input patterns are presented and the network successfully converges, the input patterns will be clustered into two categories: red and blue.

Clustering

One immediate question may be what such a clustering action accomplishes. The answer is basically the same as what a Kohonen network does (section 3.6). The network stores the input patterns through modification of weights. The weights become densest where input patterns are most common.

Suppose we observe spectral intensities of 100 stars and feed them to a network. Perhaps 20 of them are classified as reddish, 30 are bluish, and 50 are whitish. The network clusters the 100 stars into these three categories without being told by the human how many categories are expected and what these categories are. Clustering data can be applied in many domains. Medical records such as laboratory test results can be analyzed. Data for patients with a certain disease will be clustered in a different group from healthy people. Such information will help to diagnose new patients with this disease. Clustering machines based on observed characteristics may identify those that are about to fail. Clustering business firms based on their records and recent activities may reveal those that are in trouble.

Learning process

For their learning rules, there are some similarities among the Boltzmann machine and other neural network models, such as backpropagation. Initially the weights are randomly assigned, and the networks learn by adjusting the weights. The adjustments are performed over iterations, by slightly changing the weights each time by: $w_{ij}^{(new)} = w_{ij}^{(current)} + \Delta w_{ij}$. In the case of the backpropagation model, Δw_{ij} is selected to decrease the total error of the network in the steepest direction. Similarly, in the case of the Boltzmann machine, Δw_{ij} is selected to decrease the relative entropy of the network in the steepest direction. The relative entropy is a function of probabilities of the network states, and it is analogous to the one in statistical mechanics. Using these probabilities, it can be shown that the steepest direction is represented by the mean correlations between neurons. Hence, during the course of iterations these mean correlations between neurons are collected, Δw_{ij} is determined, and the weights are adjusted. This is the essence of the learning process.

Network energy

Analogous to thermodynamics, we can define the energy of the Boltzmann machine of a particular state as:

$$E = -\frac{1}{2} \sum_i \sum_j w_{ij} x_i x_j. \qquad (1)$$

In the above and the other expressions in this section, $i \neq j$ is assumed for double summation in terms of i and j, since $w_{ii} = 0$. We note that the above expression can also be written as $- \sum_i \sum_{j>i} w_{ij} x_i x_j$, by summing up for only the upper triangle for j. Suppose that we pick out a neuron x_k, and change x_k to $-x_k$ (i.e., flip +1 and -1). We want to determine ΔE_k, the change in the energy of the entire network due to such a flip. Let E' be the energy after the flip. We can consider a two-dimensional matrix whose elements are $w_{ij} x_i x_j$. The only elements affected by the flip are those in the kth row and kth column, and their summations are: $- (1/2) (- x_k) \sum_j w_{kj} x_j - (1/2)(- x_k) \sum_i w_{ik} x_i = -(- x_k) \sum_i w_{ik} x_i$. In the last step we used $w_{kj} = w_{jk}$ and replaced dummy index j with i. The counterpart term in E before the flip can be obtained by simply replacing

(- x_k) with (x_k), i.e., the term is -(x_k) Σ_i w_{ik} x_i. Hence,

$$\Delta E_k = E' - E = 2\, x_k \Sigma_i\, w_{ik}\, x_i. \tag{2}$$

(Some authors define $\Delta E_k = E - E' = -2\, x_k \Sigma_i\, w_{ik}\, x_i$. In a binary system where each neuron x_k assumes 0 or 1, we can determine the energy difference when $x_k = 1$ is changed to 0: ΔE_k = (E for $x_k = 0$) - (E for $x_k = 1$) = $\Sigma_i\, w_{ik}\, x_i$.)

State probabilities of individual neurons

During the course of the learning process, neurons are selected at random and updated according to the following probabilities. Change x_i to $-x_i$ with the probability:

$$P(x_i \rightarrow -x_i) = \frac{1}{\left(1+\exp\left(\frac{\Delta E_i}{T}\right)\right)} = \frac{1}{\left(1+\exp\left(2x_i \sum_i \frac{w_{ij}x_j}{T}\right)\right)} \tag{3}$$

where ΔE_i is the change in the energy of the entire network due to such a flip given in equation (2) and T is the current temperature for simulated annealing. x_i stays the same with the probability:

$$P(x_i \rightarrow x_i) = 1 - \mathrm{P}(x_i \rightarrow -x_i) = \frac{1}{\left(1+\exp\left(\frac{\Delta E_i}{T}\right)\right)} = \frac{1}{\left(1+\exp\left(-2x_i \sum_j \frac{w_{ij}x_j}{T}\right)\right)} \tag{4}$$

We note that the right-hand sides of equations (3) and (4) add up to 1. In equation (3), when $\Delta E_i = 0$, the probability is equal to 0.5. When $\Delta E_i > 0$, the probability is < 0.5; the probability approaches 0 as ΔE_i increases (i.e.: x_i does not flip). When $\Delta E_i < 0$, the probability is > 0.5; it approaches 1 as ΔE_i decreases (i.e.: x_i flips). These make sense in order for the network energy to decrease. The effect of temperature T is that when it is large, $|\, \Delta E_i/T \,|$ is small, so it does not produce as much of an impact on the probability of whether it will flip or not flip. When T gets small, $|\, \Delta E_i/T \,|$ becomes large, and so has a greater impact on determining whether it will flip or not flip. These features agree with the spirit of simulated annealing.

Equations (3) and (4) are consistent with the probabilities of the state of a neuron, either +1 or – 1, which is determined in stochastic way as follows:

$$x_i = \begin{cases} +1 & \text{with probability } p_i \\ -1 & \text{with probability } 1 - p_i \end{cases} \tag{5}$$

where

$$p_i = \frac{1}{1+\exp(-\frac{2}{T}\sum_j w_{ij}x_j)} \quad , 1-p_i = \frac{1}{1+\exp(\frac{2}{T}\sum_j w_{ij}x_j)} . \tag{6}$$

Again we note that the right-hand sides of the above expressions add up to 1. The consistency among equations (3) through (6) can be checked as follows. Regardless of the current x_i, i.e., x_i = 1 or -1, we see that the probability of having a new x_i = 1 is equal to p_i by applying equation (4) with x_i = 1 or (3) with x_i = -1. Similarly, we see that the probability of having a new x_i = -1 is equal to 1 - p_i, regardless of the current x_i. Alternatively, we can compute $P(x_i = 1 \text{ after update}) = P(x_i = 1 \text{ before update}) \times P(x_i \rightarrow x_i) + P(x_i = -1 \text{ before update}) \times P(x_i \rightarrow -x_i)$ and $P(x_i = -1 \text{ after update}) = P(x_i = -1 \text{ before update}) \times P(x_i \rightarrow x_i) + P(x_i = 1 \text{ before update}) \times P(x_i \rightarrow -x_i)$.

Free-running (-) and clamped (+) phases

The learning process of the Boltzmann machine includes the following two major phases:

- **Free-running phase** (also called **negative phase**, and the "-" sign is associated with this phase), where the network operates freely without the influence of the input patterns.
- **Clamped phase** (also called **positive phase**, and the "+" sign is associated with this phase), where the input patterns are clamped to network's visible neurons.

These two phases are performed alternately as -, +, -, +, ….

We label the states of the visible neurons with α, and those of hidden neurons with β. With K visible neurons, α runs from 1 to 2^K as (1, ..., 1, 1), (1, ..., 1, -1), ..., (-1, ..., -1, -1). Similarly, with L hidden neurons, β runs from 1 to 2^L. A state of the whole system of $K + L$ visible and hidden neurons is specified by an α and a β as one of 2^{K+L} possible states; we denote this specific state as $\alpha\beta$, a concatenation of α and β. Note that although the two letters α and β are used, $\alpha\beta$ represents a single state of the whole system. Let vector $\mathbf{x}_{\alpha\beta}$ represent a particular state of the network involving all the visible and hidden neurons. For example, in Fig. 3.13, $\mathbf{x}_{\alpha\beta}$ = (1, -1, -1, 1; 1, -1, 1) can be one of the 2^7 = 128 possible different states.

We place superscript - for a free-running phase and + for a clamped phase. $P(\mathbf{x}_\alpha)^-$ represents the *actual probability* of finding the visible neurons in state α at equilibrium in the free-running (-) phase. Hereafter we use simpler notation and write $P(\mathbf{x}_\alpha)^-$ as P_α^-. P_α^+ is the *desired probability* of finding the visible neurons in state α at equilibrium in the clamped (+) phase. Similarly, $P_{\alpha\beta}^-$ and $P_{\alpha\beta}^+$ represent the probabilities where the entire state is in state $\alpha\beta$ at equilibrium in a free-running and clamped phases, respectively. $P_{\beta|\alpha}^-$ is the conditional probability at equilibrium where the hidden neurons are in state β, given the visible neurons are in state α in a free-running phase. $P_{\beta|\alpha}^+$ is the same except that it is in a clamped phase.

Computing Δw_{ij} based on the relative entropy

Based on information theory, we define the relative entropy G as a measure of the difference between the distributions P_α^- and P_α^+, weighted by P_α^+, as:

$$G = \sum_{\alpha} P_{\alpha}^{+} \ln \frac{P_{\alpha}^{+}}{P_{\alpha}^{-}}. \quad (7)$$

G is always positive or zero, and is zero when $P_{\alpha}^{-} = P_{\alpha}^{+}$ for all α.

We select Δw_{ij} to decrease the relative entropy G in the direction of the steepest gradient:

$$\Delta w_{ij} = -\eta \frac{\partial G}{\partial w_{ij}}. \quad (8)$$

It can be shown that the above equation leads to (see Appendix):

$$\Delta w_{ij} = \frac{\eta}{T}\left[< x_i x_j >^{+} - < x_i x_j >^{-} \right] \quad (9)$$

where $<x_i x_j>^{+}$ and $<x_i x_j>^{-}$ are the mean correlations between neurons x_i and x_j in the clamped (+) and free-running (-) phases, respectively. The mean correlations take real values on [-1, 1], and they are determined by taking the averages of $x_i x_j$ (See Step 5 in the following Learning algorithm).

3.8.3 Unsupervised Learning by the Boltzmann Machine: Algorithms

In the following, we summarize two algorithms, a step-by-step procedure for unsupervised learning for the Boltzmann machine and a procedure for testing on the same machine (Randall C. O'Reilly, 2006, private communication). There are variations of these algorithms, and which one is best in terms of performance and computing time depends on the application. We also give two illustrative examples.

Learning (training) algorithm

The algorithm consists of sextupled nested loops.

Step 0. Initialization:

Set weights w_{ij} to random values uniformly distributed over [$-u$, u], where u is typically 0.5 or 1. Set the hidden neurons randomly to 1 or -1 with equal probability.

Step 1. Iterations over consecutive network convergences.

Repeat the rest of the algorithm until the network satisfies a convergence criterion (e.g., equation (10) below) for a certain number (e.g., 5) of consecutive times. (See Note below)

Step 2. Iterations over one-time network convergence.

Repeat the rest of the algorithm until the network converges one time. There are a few different criteria for convergence check, and the following is a common one:

$$\sum_{k} \sum_{i} \left(x_i^{-} - x_i^{+} \right)^2 < \varepsilon. \quad (10)$$

The outer summation is taken over for all input patterns, and the inner summation for all the neurons in the network, $i = 1, K + L$. x_i^- and x_i^+ are x_i values during the negative and positive phases, respectively. ε is a preset small positive number. The number of iterations of Step 2 is typically much higher than the number of distinct input patterns because of this convergence condition. When the network satisfies criterion (10), we say this is a "one-time network convergence"; back-up to Step 1.

Step 3. Iterations over input patterns.

A conclusion of the set of all input patterns is an *epoch*. After an epoch, go back to the convergence check in Step 2.

For each input pattern, perform sub-steps a, b and c, each once.

a. Positive (clamped) phase. Clamp the input pattern to the visible neurons and perform Step 4.

b. Negative (free-running) phase. Perform Step 4 without clamping the input pattern to the visible neurons.

c. Updating weights w_{ij}'s.

$$w_{ij}^{(\text{new})} = w_{ij}^{(\text{current})} + \Delta w_{ij}, \text{ where}$$
$$\Delta w_{ij} = (\eta / T_f)\,(\langle x_i x_j \rangle^+ - \langle x_i x_j \rangle^-),$$

where η is a positive constant and T_f is the final temperature in Steps 4a and 4b; $\langle x_i x_j \rangle$ is the average of $x_i x_j$ collected during Step 4b; $\langle x_i x_j \rangle^+$ is the average for the positive phase and $\langle x_i x_j \rangle^-$ is for the negative phase. (See Note below)

Step 4. Simulated annealing - iterations over temperatures.

a. For each of the clamped and free-running phases invoked in Step 3, perform simulated annealing, starting from a high temperature T_0 and gradually decreasing the temperature T. For each temperature, perform Step 5. When T reaches a small positive number T_f, the system is in thermal equilibrium at this temperature. Go to Step 4b.

b. For each of the clamped and free-running phases invoked in Step 3, perform additional iterations (say, 10 times) of Step 5, using the final temperature T_f in Step 4a. During these iterations, collect statistical data of $x_i x_j$. After the iterations, compute $\langle x_i x_j \rangle^+$ and $\langle x_i x_j \rangle^-$ necessary in Step 3c. The number of additional iterations affects the probabilistic accuracy of the collected data; the higher the number the better the accuracy.

Step 5. Iterations at each temperature T given in Step 4.

Repeat Step 6 - updating neurons x_i's, until all $|\Delta E_i|$ given in Step 6 become small enough at the temperature T. (Here "all" refers to all the neurons x_i under Items 1) or 2) in Step 6, below)

Step 6. Updating neurons, x_i's.

Perform the inner-most iterations over

1) all the visible and hidden neurons for a negative phase invoked in Step 3a, or

2) all the hidden neurons for a positive phase invoked in Step 3b.

Randomly pick out a neuron x_i and change x_i to $-x_i$ (i.e., flip +1 and -1) with the probability:

$$P(x_i \rightarrow -x_i) = \frac{1}{\left(1+\exp\left(\frac{\Delta E_i}{T}\right)\right)} = \frac{1}{\left(1+\exp\left(2x_i \sum_j \frac{w_{ij}x_j}{T}\right)\right)}$$

where ΔE_i is the change in the energy of the entire network due to such a flip and T is the current temperature set in Step 4. x_i stays the same with the probability $1 - P(x_i \rightarrow -x_i) = 1/(1 + \exp(-2\, x_i\, \Sigma_j\, w_{ji}\, x_j/T))$.

Note on Step 1.

The reason is that the network often "accidentally" gives a very small value or a zero for the criterion – a false convergence indication even if it is not actually converged. When iterations are continued further for such a case, the criterion value will return to a larger value, indicating the network was not converged. There is no scientific formula to determine the appropriate number of consecutive times in this Step. For a given problem, we can experiment by selecting a relatively large number and observing that the criterion value does not change in Step 1. The number can be affected by several factors such as the size of the network, the number of input patterns, and the degree of the complexity of each pattern. A rule of thumb is that the more complex the environment (e.g., a large network), the fewer number of consecutive convergences are required, because an accidental false convergence is less likely. (Another convergence criterion can be changes of the network weights. If all the weights do not change for a certain number (e.g., 5) of consecutive times, the network is considered as converged.)

Note on Step 3c.

Alternatively, Step 3c of updating weights can be performed after each epoch, rather than after each input pattern. Such a scheme will require significantly less computing time when there are many patterns. However, the overall performance can be less satisfactory for some problems. When weights are modified only after all patterns are

processed, delicate weight adjustments specific to each pattern may not be accomplished. This modified version can be implemented by setting up a new step, Step 3.5, between Steps 3 and 4. Steps 3a and 3b are moved to Step 3.5. New Step 3 performs iterations over patterns, invoking Step 3.5 for every pattern. After all patterns are invoked, i.e., after an epoch, perform Step 3c once.

After a network converges for a set of training (learning or exemplar) patterns employing the above Learning algorithm, new test patterns can be associated to the network. The basic idea is the same as associative memory discussed in Section 3.2. Test patterns can be generated by various ways: 1. *noisy patterns*, i.e., randomly flipping +1 and -1 for some (e.g., 25% of) pixels of one of the training patterns; 2. *completely random patterns*, i.e., picking all pixels randomly; 3. obtaining patterns from *experimental data*. Most of these test patterns should converge to their closest corresponding training patterns. However, some test patterns may converge to the wrong training patterns, or to patterns different from any of the training patterns. Such errors can occur depending on various factors such as the probability distribution of the training patterns, the quality of the test patterns and the problem complexity. The following is a possible algorithm to perform such testing.

Testing algorithm for each test pattern

Step 0. Initialization: Keep the weights w_{ij} obtained from the training session. Set the visible neurons to the testing pattern and assign values of the hidden neurons randomly.

Step 1. After the above, perform sub-step a (Negative phase) of Step 3 in the Training Algorithm. This invokes nested iterations over Steps 4a, 5, and 6. Note that in the Testing algorithm, sub-steps b. Positive phase and c. Updating weights are not performed.

Illustrative examples

My graduate students Bob Crichton and Scott Galinac performed numerical experiments on a simple Boltzmann machine, similar to Fig. 3.13. The network has 12 visible neurons and 6 hidden neurons. Three different input patterns are prepared as shown in Fig. 3.15. One pattern of (a), two patterns of (b) and three patterns of (c) are given as a set of training input patterns (i.e., the probability distribution of the input patterns is 1/6, 2/6 and 3/6, respectively). For this specific run, we set $T_0 = 1$ and (new T) = 0.90 × (current T) in Step 4 of the algorithm, to a minimum temperature of 0.05. A typical run required about 5 consecutive iterations of Step 1 for the network to converge with $\varepsilon = 0$ in Equation (10). After a successful run for network training, different test patterns are given to the network. Most of these test patterns converged to the corresponding training patterns.

(a) Blue ●●●●○○○○○○○○ (b)Yellow ○○○○●●●●○○○○ (c) Red ○○○○○○○○●●●●

Figure 3.15. Simple Boltzmann machine experiment, with 12 visible neurons and 6 hidden neurons similar to Fig. 3.13. Three types of training input patterns (a), (b) and (c) are given to the network with frequency of 1, 2 and 3, respectively (i.e., a total of six patterns). Here ● represents +1, ○ represents -1.

Bob and Scott also experimented with a larger Boltzmann machine network with 120 visible neurons and 30 hidden neurons. Three different training input patterns are prepared as shown in Fig. 3.16. As before, one pattern of (a), two patterns of (b) and three patterns of (c) are given as a set of training input patterns. A typical run required a single iteration of Step 1 for the network to converge with $\varepsilon = 0$. For the second example with 150 neurons, it did not require any number of consecutive iterations, i.e., only one-time network convergence in Step 1 was sufficient. For this specific run, we set $T_0 = 3$ and (new T) = 0.9 × (current T) in Step 4 of the algorithm, to a minimum temperature of 0.01; η in Step 3c was set to a small number, e.g., 0.0001 or 0.001.

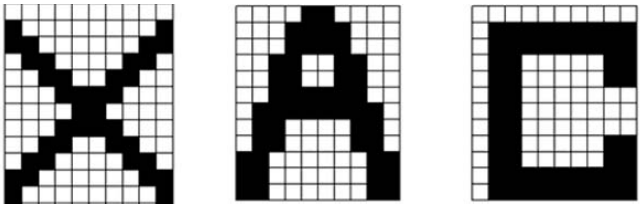


Figure 3.16. Three types of training input patterns for a Boltzmann machine with 120 visible neurons and 30 hidden neurons. The input patterns (a), (b) and (c) are given to the network with frequency of 1, 2 and 3, respectively (i.e., a total of six patterns). A black pixel represents +1, a white pixel represents -1.

After a successful run for network training, different test patterns are given to the network. They are: 1. noisy patterns (training patterns with 25% of the pixels reversed) and 2. completely random patterns. Most noisy patterns are converged correctly to their corresponding input patterns. On one test for random input patterns, 100 patterns are generated randomly and given to the network. The probability distribution of the convergence of the random patterns is close to that of the training input patterns, i.e., approximately 1/6, 2/6 and 3/6 for patterns (a), (b) and (c), respectively. Exceptions to these converged probability distributions are that when the initial temperatures are set very high (for example $T_0 = 100$ rather than 3 in Fig. 3.16 example), the distributions tend to skew toward the training pattern(s) with the highest probability distributions. For example, out of 100 completely random patterns, 75 of them may converge to Pattern (c), 5 to Pattern (b), none to Pattern (a), and the remaining to patterns different from any of the training patterns. This indicates that a selection of reasonable parameter values such as the initial

temperature and the minimum temperature is important, which may be determined through experiment. Fig. 3.17 illustrates a typical behavior of the network when it is given a noisy test pattern. For this specific run, we set $T_0 = 3$ and (new T) = 0.90 × (current T) in Step 4 of the algorithm, to a minimum temperature of 0.01.

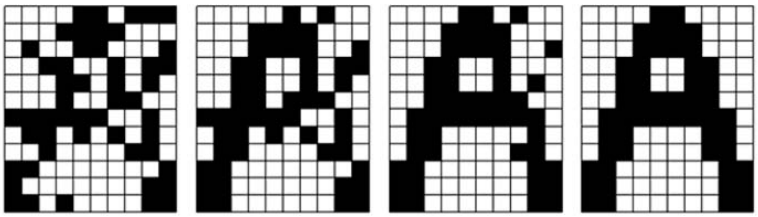


Figure 3.17. A typical behavior of the network described in Fig. 3. 16 when it is given a noisy test pattern.

3.8.4 Appendix. Derivation of Delta-Weights

(Hinton and Sejnowski, 1986, pp. 315-316)

To determine Equation (9), $\Delta w_{ij} = \frac{\eta}{T}\left[< x_i x_j >^{+} - < x_i x_j >^{-} \right]$, we substitute G given by equation (7) into equation (8), noting that P_α^{+} is independent of w_{ij} since it is the probability of the visible neurons in the clamped phase:

$$\Delta w_{ij} = -\eta \frac{\partial G}{\partial w_{ij}} = \eta \sum_{\alpha} \frac{P\alpha^{+}}{P\alpha^{-}} \frac{\partial P\alpha^{-}}{\partial w_{ij}} . \tag{11}$$

According to the Boltzmann distribution from statistical mechanics, the probability P_α^{-} is given by:

$$P_\alpha^{-} = \sum_{\beta} P_{\alpha\beta}^{-} = \frac{\sum_{\beta} \exp(-\frac{E_{\alpha\beta}}{T})}{\sum_{\lambda\mu} \exp(-\frac{E_{\lambda\mu}}{T})} \quad . \tag{12}$$

In the rightmost expression, $E_{\alpha\beta}$ is the energy of the network in state $\alpha\beta$, and is given by:

$$E_{\alpha\beta} = -\sum_{i} \sum_{j>i} w_{ij} x_i^{\alpha\beta} x_j^{\alpha\beta} \tag{13}$$

where $x_i^{\alpha\beta}$ is the ith neuron in state $\alpha\beta$. We note that $\alpha\beta$ has been subscripted in the previous expressions such as $P_{\alpha\beta}^{-}$ and $E_{\alpha\beta}$, but in $x_i^{\alpha\beta}$ it is superscripted since we need to indicate two indices, one for the entire state $\alpha\beta$ and the other for each individual neuron i. Equation (13) is the same as equation (1), except that state $\alpha\beta$ is explicitly shown and the summation is taken over the upper triangle area so that there is no 1/2

factor. The summation of the denominator of the rightmost expression in equation (12) is carried over all the states involving both visible and hidden neurons. The dummy index $\lambda\mu$ is used in place of $\alpha\beta$ to distinguish it from α and β. This denominator is a normalization factor called the partition function; we note that with this factor. $\sum_{\alpha} P_{\alpha}^{-} = 1.$

By substituting equation (13) into equation (12), P_{α}^{-} is expressed in terms of w_{ij} and is directly differentiable with respect to w_{ij}. The derivative will be used for the last factor of equation (11). To carry out the differentiation, we first evaluate

$$\frac{\partial}{\partial w_{ij}}\{\exp(-\frac{E_{\alpha\beta}}{T})\} = \exp(-\frac{E_{\alpha\beta}}{T})\frac{\partial}{\partial w_{ij}}(-\frac{1}{T}[-\sum_{i}\sum_{j>i} w_{ij}x_i^{\alpha\beta}x_j^{\alpha\beta}]) = \frac{1}{T}x_i^{\alpha\beta}x_j^{\alpha\beta}\exp(-\frac{E_{\alpha\beta}}{T}) \quad (14)$$

In the above, from the first expression to the second, we substituted equation (13) into the second occurrence of $E_{\alpha\beta}$. From the second expression to the third, we note that all except the w_{ij} term are zero when differentiated with respect to w_{ij}. We now compute $\frac{\partial P_{\alpha}^{-}}{\partial w_{ij}}$ by equation (12), by applying the calculus formula $(u/v)' = u'/v - uv'/v^2$, and by using the result of equation (14):

$$\frac{\partial P_{\alpha}^{-}}{\partial w_{ij}} = \frac{\frac{1}{T}\sum_{\beta} x_i^{\alpha\beta}x_j^{\alpha\beta}\exp(-\frac{E_{\alpha\beta}}{T})}{\sum_{\lambda\mu}\exp(-\frac{E_{\lambda\mu}}{T})} - \frac{\left[\sum_{\beta}\exp(-\frac{E_{\alpha\beta}}{T})\right]\left[\frac{1}{T}\sum_{\lambda\mu} x_i^{\lambda\mu}x_j^{\lambda\mu}\exp(-\frac{E_{\lambda\mu}}{T})\right]}{\left[\sum_{\lambda\mu}\exp(-\frac{E_{\lambda\mu}}{T})\right]^2}$$

$$= \frac{1}{T}\left[\sum_{\beta} P_{\alpha\beta}^{-}x_i^{\alpha\beta}x_j^{\alpha\beta} - P_{\alpha}^{-}\sum_{\lambda\mu} P_{\lambda\mu}^{-}x_i^{\lambda\mu}x_j^{\lambda\mu}\right]. \quad (15)$$

Substituting equation (15) into equation (11) yields:

$$\Delta w_{ij} = -\eta\frac{\partial G}{\partial w_{ij}} = \eta\sum_{\alpha}\frac{P_{\alpha}^{+}}{P_{\alpha}^{-}}\frac{\partial P_{\alpha}^{-}}{\partial w_{ij}}$$

$$= \frac{\eta}{T}\sum_{\alpha}\frac{P_{\alpha}^{+}}{P_{\alpha}^{-}}\left[\sum_{\beta} P_{\alpha\beta}^{-}x_i^{\alpha\beta}x_j^{\alpha\beta} - P_{\alpha}^{-}\sum_{\lambda\mu} P_{\lambda\mu}^{-}x_i^{\lambda\mu}x_j^{\lambda\mu}\right]$$

$$= \frac{\eta}{T}\left[\sum_{\alpha}\frac{P_{\alpha}^{+}}{P_{\alpha}^{-}}\sum_{\beta} P_{\alpha\beta}^{-}x_i^{\alpha\beta}x_j^{\alpha\beta} - (\sum_{\alpha} P_{\alpha}^{+})(\sum_{\lambda\mu} P_{\lambda\mu}^{-}x_i^{\lambda\mu}x_j^{\lambda\mu})\right]. \quad (16)$$

Based on probability theory we have:

$$P_{\alpha\beta}^{+} = P_{\beta|\alpha}^{+}P_{\alpha}^{+} \text{ and } P_{\alpha\beta}^{-} = P_{\beta|\alpha}^{-}P_{\alpha}^{-}. \quad (17)$$

Also, in equilibrium are,

$$P_{\beta|\alpha}^{-} = P_{\beta|\alpha}^{+}. \quad (18)$$

Since these are conditional probabilities representing the hidden neurons are in state β given the visible neurons are in state α, they must be the same whether the visible neurons are clamped or not. Using equations (17) and (18) we have:

$$\frac{P_\alpha^+}{P_\alpha^-} P_{\alpha\beta}^- = P_{\alpha\beta}^+. \tag{19}$$

Based on probability theory we also have:

$$\sum_\alpha P_\alpha^+ = 1. \tag{20}$$

Substituting equations (19) and (20) into equation (16) we have:

$$\Delta w_{ij} = \frac{\eta}{T}\left[\sum_{\alpha\beta} P_{\alpha\beta}^+ x_i^{\alpha\beta} x_j^{\alpha\beta} - \sum_{\lambda\mu} P_{\lambda\mu}^- x_i^{\lambda\mu} x_j^{\lambda\mu}\right]. \tag{21}$$

In general, multiplying any quantity Φ_i for each event i by its corresponding probability P_i and summing up the products for all events gives $<\Phi>$, the expected or average value of Φ_i. The above is a special case where $\Phi = x_i x_j$. Multiplying x_i and x_j together with their corresponding probability for each state, and summing up the products over all possible states, gives the mean correlations between neurons x_i and x_j that can be denoted as $<x_i x_j>$. Using this notation, the first term in the brackets is $<x_i x_j>^+$, the mean correlations in the clamped (+) phase. Similarly, the second term is $<x_i x_j>^-$, the mean correlations in the free-running (-) phase. Hence,

$$\Delta w_{ij} = \frac{\eta}{T}\left[< x_i x_j >^+ - < x_i x_j >^-\right]. \tag{22}$$

Further Reading

In addition to the literature cited at the end of Chapter 2, the following books and articles discuss specific topics as described below.

The following book presents many optimization problems solved by applying the Hopfield-Tank model.

Y. Takefuji, *Neural Network Parallel Computing*, Kluwer Academic, 1992.

The following three are seminal articles of the Hopfield and Hopfield-Tank models.

J.J. Hopfield, "Neural Networks and Physical Systems with Emergent Collective Computational Abilities," *Proceedings of the National Academy of Sciences*, Vol. 79, 1982, 2554-2558.

J.J. Hopfield, "Neurons with Graded Response Have Collective Computational Properties like Those of Tow-state Neurons," *Proceedings of the National Academy of Sciences*, Vol. 81, 1984, 3088-3092.

J.J. Hopfield and D.W. Tank, "'Neural' Computation of Decisions in Optimization Problems," *Biological Cybernetics*, Vol., 52, 1985, 141-152.

The following two discuss the Kohonen models.

T. Kohonen, "The 'Neural' Phonetic Typewriter," *Computer*, Vol. 21, 3, 1988, 11-22.

T. Kohonen, *Self-Organization and Associative Memory*, 3rd Ed., Springer-Verlag, 1989.

The following two are references for simulated annealing.

S. Kirkparick, C.D. Gelatt, Jr., and M.P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, 1983, pp. 671-680. Reprinted in J.A. Anderson and E. Rosenfeld, Eds., *Neurocomputing: Foundations of Research*, MIT Press, 1988.

S. Geman and D. Geman, "Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6, 1984, pp. 721-741.

The following three are seminal articles on Boltzmann machines.

G. E. Hinton and T. J. Sejnowski, " Optimal Perceptual Inference," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Washington, DC, New York: IEEE, 1983, pp. 448-453.

D. H. Ackley, G. E. Hinton, and T. J. Sejnowski, "A Learning Algorithm for Boltzmann Machines," *Cognitive Science*, vol. 9, 1985, pp. 147-169. Reprinted in J.A. Anderson and E. Rosenfeld, EDs., *Neurocomputing: Foundations of Research*, MIT Press, 1988.

G. E. Hinton and T. J. Sejnowski, "Learning and Relearning in Boltzmann Machines," in D.E. Rumelhart, J.L. McClelland and the PDP Research Group (Eds.), *Parallel Distributed Processing*, Vol. 1, MIT Press, 1986, pp. 282-317.

The following two cited in Chapter 2 provide tutorials for Boltzmann machines and other neural network models.

J. Hertz, A. Krogh and R.G. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley, 1991.

S. Haykin, *Neural Networks: A Comprehensive Foundation*, 2nd Ed., Prentice-Hall, 1999.

4 Genetic Algorithms and Evolutionary Computing

4.1 What are Genetic Algorithms and Evolutionary Computing?

During the four billion year history of the earth, biological life was born, perhaps as a result of a series of rare chance chemical and physical reactions of molecules. Over time, more and more complex forms of biological life evolved. **Genetic algorithms** are computer models based on genetics and evolution in biology. The basic elements of a genetic algorithm are: *selection* of solutions based on their goodness, *reproduction* for crossover of genes, and *mutation* for random change of genes. Through these processes, genetic algorithms find better and better solutions to a problem just as species evolve to better adapt to their environments.

Genetic algorithms have been extended in their ways of representing solutions and performing basic processes. A broader definition of genetic algorithms, sometimes called **evolutionary computing**, includes not only generic genetic algorithms but also classifier systems, genetic programming where each solution is a computer program, and some aspects of artificial life. Other related areas include evolvable hardware, evolutionary robotics, ant colony optimization, and swarm intelligence.

Genetics in real life

Before studying genetic algorithms, we will briefly review genetics in real life, e.g., in human. A life of a human body starts at fertilization of an egg by a sperm. Before conception, there are 23 **chromosomes**, numbered No. 1, 2, ..., 23 in an egg, and similarly, 23 numbered chromosomes in a sperm (a total of 46). In a diploid organism like human, two chromosomes of the same number, one from the egg and the other from the sperm, make a pair of chromosomes for the child (e.g., chromosomes mother No. 1 and father No. 1 make child pair No. 1) (Fig. 4.1).

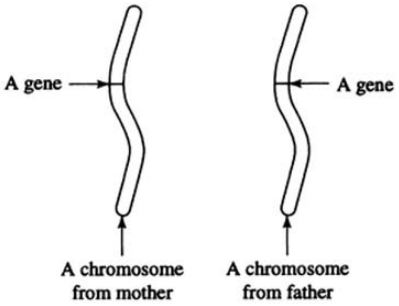


Fig. 4.1A pair of chromosomes for a child

When we closely look at a pair of chromosomes for the child, there are thousands or even millions of **genes** on each chromosome. A gene can be thought of as a tiny point on a chromosome. One gene from the mother and the corresponding gene from the father make a gene-pair for the child. Each pair (or a certain number of pairs) of genes contributes to specific characteristics of the child, such as the blood type, color of eyes, etc. Such characteristics are called **phenotypes**.

As an illustration, let us see how the child's blood type is determined as one of four possible types, *O*, *A*, *B*, or *AB*. Each gene for blood type can have a value of either 0, 1, or 2, called **alleles**. Possible gene-pair combinations, called **genotypes**, and their corresponding phenotypes are:

Genotype	Phenotype
00	O
10 or 11	A
20 or 22	B
21	AB

For example, if the gene value from mother is 0 and from father is 1, the child's genotype will be 10 and the phenotype will be blood type *A*. Note that the order of gene values in real life is immaterial (e.g., 10 = 01). Alleles differ for different kinds of genes (e.g., a specific gene may have either 0 or 1, another gene may have 0, 1, 2, or 3, and so on).

When this child grows to an adult and produces an egg (if female) or a sperm (if male), one chromosome from each chromosome pair is selected for the egg or sperm. This is how certain phenotypes are inherited from a person to a child, from the child to a grandchild, and so on. Most gene values are inherited from mother and father to the child as they are. Occasionally, however, some gene values change (e.g., from 0 to 1), perhaps because some unusual physical, chemical, or biological effects (e.g., a gene is hit by a cosmic ray, etc.). Such a change of a gene value is called **mutation**.

In any species in biology, those individuals who better adapt to the

environment have higher probabilities for survival, thus they have higher probabilities for producing their offspring. Over generations, this process is repeated, and the result is that those individuals and genes that better adapt to the environment tend to remain while those that don't tend to disappear, i.e., become extinct. This theory of a natural screening process is called **(Darwinian) evolution**.

The basic idea of genetic algorithms

The computer genetic algorithms which we will study are abstract models of natural genetics and the evolution process discussed above. Genetic algorithms include concepts such as chromosomes, genes, mating or crossover breeding, mutation, and evolution. We will not, however, attempt to build computer models as close as possible to natural genetics. Rather, we will develop useful models that are easy to implement in computers by borrowing concepts from natural genetics.

The major process of our genetic algorithm is as follows. At the beginning, we randomly generate solutions or "chromosomes" for the problem. After the initial random generation of solutions, we perform iterations. Each iteration consists of several steps - we select good solutions and perform crossover breeding; occasionally we may have mutations on certain solutions. Through selection of good solutions during iterations, the computer will develop increasingly better solutions as in the case of natural evolution. We can apply this approach to many types of problems such as optimization and machine learning.

4.2 Fundamentals of Genetic Algorithms

Representations of solutions

A genetic algorithm starts with designing a representation of a solution for the given problem. A **solution** here means any value that is a candidate for a **correct solution** or a final answer; a solution may or may not be the correct solution. For example, suppose we want to maximize function $y = 5 - (x - 3)^2$. Then $x = 1$ is a solution, $x = 2.5$ is another solution, and $x = 3$ is the correct solution that maximizes y.

The representation of each solution for a genetic algorithm is up to us. It depends on what each solution looks like and what solution form will be convenient for applying a genetic algorithm. The most common representation of a solution is a string of characters.

Consider a finite-length **string** of characters over a fixed alphabet, e.g., {0,1}, {0, 1, 2}, {0, 1, *}, {0, 1, 2, ..., 9}, or {*A*, *B*, ..., *Z*}, etc. We then choose the length of each string, such as, 12, 64, or 256, depending on the alphabet used and the amount of information we want to represent in each string. The larger the alphabet the more information can be represented by each character; therefore, fewer characters are necessary to encode a specific amount of information. A string is somewhat analogous to a chromosome or a set of chromosomes. Suppose that we represent each solution by a 12-bit string over the alphabet {0, 1}. A solution in this case may represent a set of values of 12 variables or parameters, each bit

representing a binary value of a parameter. Each parameter, i.e., a bit in this case, is analogous to a gene. Or, the range of each parameter may be larger than binary 0 and 1. A solution of 12 bits may represent values of 3 parameters, each parameter using 4 bits. In this case, each parameter can range binary 0000 to 1111, or decimal 0 to 15; each solution or chromosome has 3 genes.

Given an application problem, we can represent each solution as a fixed-length string, say, 32 bits. For example, a company is making four kinds of products and the problem is to find the number of products to make in order to maximize the profit under certain conditions. Then specific amounts of the products, e.g., (30, 10, 25, 40) for Products 1, 2, 3 and 4, respectively, is a solution, (20, 20, 30, 35) is another solution, and so on. We can represent a solution by a string, assigning the first 8 bits to represent the amount for Product 1, the next 8 bits for Product No. 2, and so on, with the total of 32 bits.

As said before, the representation of each solution for a genetic algorithm is up to us. Although string representation of a solution is common, other forms of representation may be more convenient for other problems. For example, for certain graph problems, a graph can be a solution. A graph can be represented by an adjacency matrix for certain problems. For a weighted graph problem, each solution may be a matrix whose elements represent the weights associated with the edges. For genetic programming problems, each solution is a computer program, much more structured than a character string. We should be flexible for adopting the most appropriate form of solution representation for each problem. In this and following sections, we will use simple string representation on the alphabet, {0, 1}.

The fitness of a solution

The **fitness** of a solution is a measure that can be used to compare solutions to determine which is better. For example, a company is trying to maximize a profit. The profit itself can be used as the fitness, or a scaled value of the profit can be the fitness. In the following we will briefly describe the fundamental steps of a genetic algorithm. The meaning of these steps will become clearer when we see examples in the following sections.

Basic steps of a genetic algorithm

There are variations and extensions of the genetic algorithm procedure. The following is a simple and typical one. A set of solutions at a specific time step is called the **population**.

Step 0. *Initialization of the population.*

Generate a set of solutions randomly.

Repeat the following three steps until the correct (optimal) solution is found, or more generally, until a terminal condition is satisfied. For certain problems, we may not know the correct solution. In such a case, we set up terminal condition(s). For example, we keep track of the best solution in each iteration. When it does not improve over a certain number of iterations (e.g., 10), we terminate the iterations.

Step 1. *Reproduction*

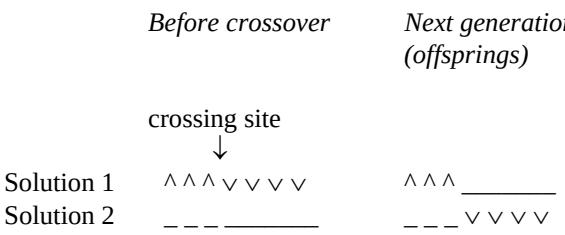
(a) Determine the fitness values and their corresponding probabilities for all the solutions in the population.

(b) Creation of a **mating pool**. Randomly select solutions weighted by the fitness. Solutions with higher fitness are more likely to be picked out than the unfit ones and tend to survive into the next generation. Here the evolution concept based on the principle of natural selection is employed.

Step 2. *Crossover (recombination) breeding*

(a) Take two solutions randomly at a time. With a fixed crossover probability p_c (e.g., $p_c = 0.7$), randomly determine whether crossover takes place. If crossover does take place, go to the next substep (b); otherwise, form two offspring that are exact copies of the two solutions (parents), and go to Step 3.

(b) Select randomly internal points (**crossing sites**) of the solutions, then swap the solution parts that follow these points.



Perform this Step 2 for all the solutions obtained in Step 1, i.e., until the new population size reaches the initially set population size, randomly selecting a pair at a time.

The significance of the crossover operations is as follows. Each solution may represent a set of values for parameters, or a prescription for performing a particular task, and so on. Parts of each solution (e.g., substrings of a solution string) may contain notions of importance or relevance. A genetic algorithm exploits this information by reproducing high quality notions, then crossing over these notions among high performers.

Step 3. *Random mutation* (or simply *mutation*).

With a certain fixed small mutation probability, p_m (e.g., $p_m = 0.001$), randomly select a small portion of the solutions and artificially change it (e.g., a bit of 1 to 0 or 0 to 1). The frequency of mutation is typically small (e.g., one mutation per thousand bit transfers).

The idea of Step 3 is again modeled from natural mutation. In this way,

we expect to create a new breed which would not be possible from the ordinary reproduction and crossover breeding processes. In genetic algorithms, after a certain number of iterations, sometimes most solutions in the population become alike so that no further significant changes occur, yet they are far from optimal. The parts changed by mutation often shake the set of solutions out of such a static configuration.

The above procedure constitutes one computer run. Often we repeat a certain number of (e.g., 10) runs starting with different random number seeds, then find the best solution among them. This is because a genetic algorithm does not guarantee the optimal solution.

4.3 A Simple Illustration of Genetic Algorithms

To better understand the fundamentals of the genetic algorithm discussed in the previous section, we will consider a scenario to which the algorithm may be applied. Then we will apply the algorithm step by step to a fictitiously simple example.

A sample scenario for a genetic algorithm application

We will describe a sample scenario to which the genetic algorithm may be applied. The key point is that different looking problems in various application areas often reduce to the same problem, thus allowing applications of the same algorithm. Typical situations where genetic algorithms are particularly useful are in difficult cases for which analytical methods do not work well.

Job shop scheduling

A *job shop* is a system composed of a series of work stations capable of making products. "Products" can be tangible goods as well as intangible items such as services. Our objective is to determine an optimal schedule which maximizes the profit of the entire shop by, for example: making an appropriate amount of products; reducing costs of production, idle time, and inventory; and avoiding possible penalties resulting from tardiness (i.e., not meeting the completion deadline) and low product quality. Obviously, the problem is important for a business's competitiveness or even for its survival. The scenario applies to situations such as the well known "just-in-time" systems. The problem is difficult, since there are not only many complex regular factors, but also irregular ones, such as sudden machine breakdowns, delays of supplies, etc. The schedule must be able to adjust flexibly and quickly to these unpredictable irregular situations. Because of these complexities, the problem cannot be expressed in an elegant mathematical formula such as linear programming.

There are many variations of the problem, depending on how its characteristics are defined (e.g., how many different kinds of products are made at the job shop,

which operations on which products are performed at each work station, the profits, costs, and constraints for making these products at each work station, and so on). An extension is a system of multi-job shops, where certain job shops are interrelated. A system of a manufacturer, suppliers of raw materials and energy, and wholesalers would be an example of an extension.

Here we consider a simple scenario of a job shop with three work stations, 1, 2, and 3. At each work station, only one kind of product is made. From running the job shop, we may observe that a schedule yields a total profit of 83 by making 10, 15, 12 units of the products at Work Stations 1, 2, and 3, respectively. Another run under the same circumstances may find a solution of 11, 17, 12 products yields the total profit of 91, and so on. We further simplify the problem and assume that each work station either makes a certain number of the product (this case can be represented by a binary 1), or does not make (represented by a binary 0). A schedule will then be represented by a three-bit string. For example, 101 means that (make, not make, make) at Work Stations 1, 2, and 3, respectively. This particular schedule may yield a total profit of 5. Another schedule (001) may be observed having a profit of 1.

A simple, step by step illustration of the genetic algorithm

Representation of solutions

Assume that each solution is represented by a three-bit string, and the population size n is 4. Assume also that the fitness value of each string is merely the binary number represented by the string as a whole (e.g., a three-bit string 101 gives the fitness value of decimal 5). Our problem is to find the string that gives the maximum fitness value, i.e., string 111 for the maximum fitness value of 7. In real-world situations, of course, the fitness values are not determined in such a simple manner. The fitness values may be determined from complex formulas, simulation models, or by referring to observations from experiments or real problem settings. Our objective here is to illustrate the basic steps of the genetic algorithm, which can be applied to real-world problems, assuming that their fitness values are determined appropriately.

Step 0. Initialization of the population

Using random numbers, suppose that the following initial population is generated. Observe that each solution is a three-bit string and the population size n is 4.

101
001
010
110

Iteration 1, Step 1. Reproduction - generation of a new mating pool

(a) Determine the fitness value for each solution, i.e., string, A_i, and the **total fitness**, $F = \Sigma f_i$.

i	A_i	f_i
1	101	5
2	001	1
3	010	2
4	110	6
		F =14

Compute the **fitness probability** (or **normalized fitness**), $p_i = f_i / F = f_i / \Sigma f_i$, and **expected count** = $n \cdot p_i$.

i	A_i	f_i	p_i	$n \cdot p_i$
1	101	5	0.357	1.429
2	001	1	0.071	0.286
3	010	2	0.143	0.571
4	110	6	0.429	1.714
	Total	14	1.000	4.000
	Avg	3.5	0.250	1.000
	Max	6	0.429	1.714

(b) Randomly select a new set of n strings (mating pool) whose *average distribution is equal to the expected count distribution*. In the above example, out of a new set of 4 strings, 1.714 strings of No. 4 will be selected as an average, 1.429 strings of No. 1 will be selected as an average, and so on. Since the number of strings must be a whole number, the actual number of strings for a particular reproduction will not contain a fraction. The most likely actual number of strings generated for No. 4 would be 2, which is the closest to 1.714; the number can be 1; the number can also be either 0, 3, or 4, but the chances for these numbers are much smaller. More precisely speaking, the *expected number* of String No. 4 out of 4 strings is 1.714. Suppose that we generated 1,000 sets of 4 strings, i.e., 4,000 strings total, then about 1,714 strings will be String No.4. When we randomly pick a set of 4 strings from this big pool of 4,000 strings, a specific set may contain 0, 1, 2, 3, or 4 strings of String No. 4. String No. 2 is likely to disappear from the new mating pool since its expected count 0.286 is small.

This process of creating a set of n strings can be implemented by assigning the ranges of random numbers to represent the probabilities of the strings being included. We choose the distribution of the random numbers to be proportional to the probability distribution of the strings. (This technique is common in the Monte Carlo method - a simulation technique using random numbers).

In our example, we can pick out a three-digit random number to match the three-digit probability to select a string. We may assign a random number to string No. *i* as follows.

Random No.	Probability	String No. *i*
000 - 356	0.357	1
357 - 427	0.071	2
428 - 570	0.143	3
571 - 999	0.429	4

There are 1,000 three-digit random numbers 000 through 999 in the above example. Each of these random numbers has an equal probability of 0.001 to be picked out. There are 357 random numbers 000 through 356. Hence, the probability of picking out one of the 357 random numbers, 000 through 356, is 0.357. If we want a better accuracy for the probability, then we can use more digits, as for example, 0.3571 for String No. 1, and so on. We would then use four digit random numbers, as for example, 0000 - 3570 to represent No. 1. Suppose we select a three-digit random number as if picking out a lottery number and it is 652 (a number in the range of 571 - 999); then we select String No. 4, "110". We repeat this process for *n* strings.

Suppose that in our experiment, we generated $n = 4$ random numbers, 483, 091, 652, and 725, and selected strings accordingly as No. 3, 1, 4, and 4 in this order. The tally is as follows.

i	A_i	np_i Expected Count	Actual Count from Experiment
1	101	1.429	1
2	001	0.286	0
3	010	0.571	1
4	110	1.714	2

The newly generated mating pool is:

(New) I	A_i
1	010
2	101
3	110
4	110

Iteration 1, Step 2. Crossover breeding

The following substeps are performed for all the solutions (strings), two solutions at a time.

(a) A pair is selected at random for mating. We determine if crossover takes place based on a pre-fixed crossover probability. In the following, we assume that crossover takes place.

(b) A crossing site is chosen uniformly at random over the length of each string. If the length is L, and the crossing site is bit position k, k is chosen over [1, L-1]. For example, when L is 3, k is chosen over [1, 2]. Suppose a string of L = 3 is 101, then bit position k is defined as follows (":" indicates the crossing site):

Bit position k:	1 ↓	2 ↓
String example	1 :	0 : 1

The solution parts of the two solutions are swapped at the crossing site.

For example, in the above, strings No. 2 and No. 3 may be selected randomly as Substep (a):

A_2	101
A_3	110

Then k = 2 may be randomly chosen as Substep (b):

A_2	10 : 1
A_3	11 : 0

Crossover takes place, generating two new strings, A'_2 and A'_3 as Substep (b):

A'_2	100
A'_3	111

The remaining two strings are mated as Substep (a), with k = 1 as a random crossing site, generating two new strings A'_1 = 010 and A'_4 = 110 as Substep (b). (In this example A'_1 and A'_4 happen to be equal to A_1 and A_4, respectively, since both A_1 and A_4 have the same last substring 10. In general, A'_1 and A'_4 are different from A_1 and A_4.) The following shows the result of the crossover process.

i	A_i	Mate No.	Crossover Site, k	New Population, A'_i
1	010	4	1	010
2	101	3	2	100
3	110	2	2	111
4	110	1	1	110

Iteration 1, Step 3. Random mutation

Mutation occurs infrequently, for example, one mutation per thousand bit transfers. In the above example, only 12 bits are transferred. Hence, mutation is very unlikely to occur for this iteration (the probability is 12/1000 = 0.012). If, however, we had unusually high mutation rate of one per, say, 12 bit transfers, we could have a mutation. By a mutation, for example, the right-most digit of A'_2 may be randomly picked out, and the string may change its value from 100 to 101.

We repeat the above iteration processes - computation of fitness and reproduction, crossover breeding, and possible mutation.

Iteration 2, Step 1 (a part)

i	A_i	f_i
1	010	2
2	100	4
3	111	7
4	110	6
Total		19
Ave		4.75
Max		7

We see an overall improvement from the initial population to this population on the Total, Ave, and Max. In this specific example, since the optimal solution for the maximum fitness value 7 is already obtained, no further iteration is necessary. In general, we would further repeat the process; usually the solutions get better and better for every iteration, and eventually and hopefully we will find the optimal solution.

4.4 A Machine Learning Example: Input-to-Output Mapping

In the previous section, we studied the basic steps of the genetic algorithm using a simple example. The example was an optimization problem, one of the most common application categories of genetic algorithms. In general, an optimization problem attempts to determine a solution that, for example, maximizes a profit or minimizes a cost. In this section, we will see another simple example. The objectives of this section are twofold: one, to better understand the basic steps of genetic algorithms by repeating a similar process; two, to see another typical application category of genetic algorithms, called *machine learning* in the domain of **input-to-output mapping**.

The basic idea of input-to-output mapping is to come up with an appropriate form of a function, which is typically simpler than the given original mapping. Incidentally, the terms "function" and "mapping" are synonyms and they are used interchangeably. In general, a function from set A to set B assigns a unique element

$f(a) = b \in B$ to each element $a \in A$. The element a is called an argument of the function f, and b is called the image of a. Set A is called the domain and set B the range of the function.

In our example, suppose that there are n input units, x_1, ..., x_n, and m output units, y_1, ..., y_m. As a simple case, assume that each of the units can take either a value of 0 or 1. Extensions of this will be that each unit can take one of the values of {0, 1, 2, ..., 9}, or, a continuous real value between -1 and 1, etc. Each input pattern, or simply input (for example, $x_1 = 0$, $x_2 = 1$, ..., $x_n = 0$, etc.) is an argument of a mapping. For each input, we are given an image, which is an output pattern or simply output (for example, $y_1 = 1$, $y_2 = 0$, ..., $y_m = 1$, etc.). Since each of x_1, ..., x_n, can take one of two possible values, 0 or 1, there are a total of $2 \times 2 \times ... \times 2 = 2^n$ input patterns, that is the size of the function domain is 2^n.

In certain application cases, all of the 2^n input patterns are given corresponding output patterns. In other cases, some input patterns may not give corresponding output patterns, that is, the mapping information is incomplete. In either case, our objective is to determine an appropriate function to best describe the given mapping. Measures of the term "best" depend on a specific application. Common measures are the accuracy of the function, the robustness and computational efficiency. Generally, determining a function that satisfies all these criteria is not necessarily an easy task, depending on the complexity of the problem. If, however, a good function is determined, it can be used for many types of applications. Since functions are defined in very general terms, they have many types of applications. Similar discussions are given before in Section 2.8.

One application category of functions is **pattern classification**. For example, an input pattern can be a two-dimensional visual image such as given in Fig. 2.4. Each small square is represented by x_i, and its value is 1 if a part of the pattern is within the square, 0 otherwise. In this example, output can be only one, y; its value is 1 if the input pattern is recognized as character "*A*", 0 otherwise. We can extend the output to, for example, y_1 and y_2. y_1 will be 1 if "*A*"; 0 if other than "*A*". y_2 will be 1 if the input pattern is "*B*"; 0 if other than "*B*". This type of application is called pattern classification since input patterns are classified into "*A*," "*B*," etc. Input patterns can be other than two-dimensional images; for example, they can originate from acoustic, or other physical and chemical measurements, and so on.

Another application category is **control**. For example, input may be temperature and humidity measurements at various points in a building. Output may be amounts of heat and humidity sources to be applied to various points in the building. Once the function is determined, it can be used to efficiently control the comfortability of the building. This concept can be applied to other control problems such as for cars, appliances, plants, and so on.

A third category is **prediction**. Given the current and recent data as input, we want to determine a most likely future outcome as output. If we are successful in implementing input-to-output mapping from a past experience, the function can be used for prediction. For example, by placing data before many breakdowns of a machine, we may be able to predict future breakdowns thereby avoiding costly shutdowns and repairs.

The automatic development of a function by the computer described here is considered as one type of machine learning in AI. The concept of machine learning

is that the computer gets smarter by itself, that is, automatic acquisition and discovery of knowledge from data and experience. In the following, we will see a simple example of a mapping. Our pace will be faster than in the previous section, since we have already understood basic steps of genetic algorithms.

Problem description

As a special case of a mapping described above, consider three input units, x_1, x_2, and x_3, and one output unit y. Each unit can take a value of either 0 or 1. Our specific target mapping is given as follows (y_t stands for target y):

x_1	x_2	x_3	y_t
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

For example, the second line of the table says that when $x_1 = 0$, $x_2 = 0$ and $x_3 = 1$, the output y should be 0. We note that the complete information of $2^3 = 8$ images are given in this example.

Representation of the function

We would like to develop our function as follows. We introduce weights, denoted as w_1, w_2, and w_3, where each w_i can be either -1, 0 or 1. Given x_1, x_2 and x_3, we first compute weighted sum, s, as: $s = w_1x_1 + w_2x_2 + w_3x_3$. We then determine computed y, y_c, as: $y_c = 0$ if $s < 0$; $y_c = 1$ otherwise. A graphical interpretation of our method is illustrated in Fig. 4.2. (If you have already studied Chapter 2, you see this is a special case of a simple perceptron discussed in Section 2.7.)

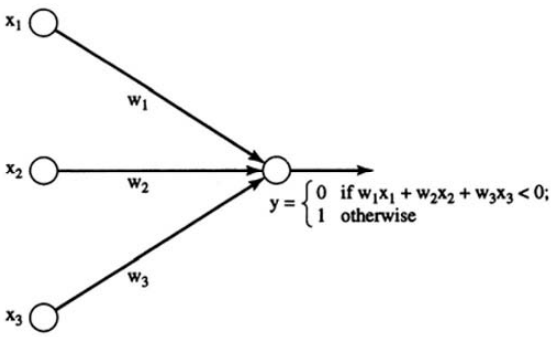


Fig. 4.2 A graphical interpretation of our functional structure for (x_1, x_2, x_3) to y.

Some students might wonder why not use the original table for our objective rather than this somewhat complicated method we are about to explore. For our simple example of only eight table entries, the original table can be used. For a larger problem, the table form is not practical. For example, if there are 10 input units, then even if we still assume that each unit can take either 0 or 1, and only one output unit y, there will be $2^{10} \approx 10^3 = 1000$ table entries. If there are 100 input units, then there will be $2^{100} \approx 10^{30}$ table entries. Our weighted sum method and its extensions may allow us to represent a function in a more concise way than using the original table.

Genetic algorithm strategy

Our problem is to determine values of w_1, w_2, and w_3 in such a way that the resulting function is as close to the original table, hopefully achieving a perfect match. In our genetic algorithm, a natural choice of each solution is a set of values of w_1, w_2, and w_3. For example, (w_1, w_2, w_3) = (-1, 0, 1) is a solution. If we want to use a binary string to represent a solution, we could use two bits for each of -1, 0 and 1, as for example: 00 for 0, 01 for 1, 11 for -1, and 10 not used. In this representation, the solution (-1, 0, 1) will be 110001. We would treat two bits as an inseparable string unit, so that a crossing site does not cut a unit into two halves. However, this bit representation is not necessary, and we will use -1, 0 and 1 in the following.

Given a solution, we can compute y_c and see how they match y_t, the target y. For example, for solution (-1, 0, 1), we have the following.

x_1	x_2	x_3	y_c	y_t	$y_c = y_t$?
0	0	0	1	1	yes
0	0	1	1	0	no
0	1	0	1	1	yes
0	1	1	1	0	no
1	0	0	0	1	no
1	0	1	1	1	yes
1	1	0	0	1	no
1	1	1	1	1	yes
					4 yes

For example, the second line of the above is computed as: $s = -1 \times 0 + 0 \times 0 + 1 \times 1 = 1$; hence $y_c = 1$.

The greater the number of "yeses," the better the solution. Hence, let the number of "yeses" be our fitness, f. For example, the fitness of the above solution (w_1, w_2, w_3) = (-1, 0, 1) is 4. A solution whose fitness is 8 represents perfect match for all the 8 cases of $y_c = y_t$, that is, a correct solution.

Applying genetic algorithm

Step 0. Initialization of the population

Using random numbers, suppose that the following initial population of size 4 is generated.

w_1	w_2	w_3
-1	0	1
1	0	0
0	1	-1
1	-1	1

Iteration 1. Step 1. Reproduction - generation of a new mating pool

The fitness values, *f*, are determined as follows by counting the number of yeses for each solution for the current population.

w_1	w_2	w_3	f
-1	0	1	4
-1	0	0	2
0	1	-1	6
1	-1	1	5
			17

Assume that the fitness probability for each solution is computed. Then the following solutions are randomly generated according to the fitness probability distribution. Note that the second solution in the above with $f = 2$ has disappeared, while the third solution with $f = 6$ is generated twice in the following.

w_1	w_2	w_3
-1	0	1
0	1	-1
0	1	-1
1	-1	1

Iteration 1. Step 2. Crossover breeding

Suppose that the first two and last two solutions are randomly mated. Then their crossing sites are randomly chosen as indicated. We assume that there is no mutation.

w_1	w_2	w_3
-1	0	1
0	1	-1
0	1	-1
1	-1	1

Iteration 2. New population.

After the above crossover breeding, the new population is:

w_1	w_2	w_3
-1	0	-1
0	1	1
0	-1	1
1	1	-1

Iteration 2. Step 1. Reproduction.

w_1	w_2	w_3	f
-1	0	-1	4
0	1	1	6
0	-1	1	4
1	1	-1	7
			21

Based on the fitness probability distribution, suppose that the following solutions are randomly generated. We note that all the above solutions are regenerated in a different order in this specific run.

w_1	w_2	w_3
0	1	1
0	-1	1
-1	0	-1
1	1	-1

Iteration 2. Step 2. Crossover breeding

Suppose that the first two and last two solutions are randomly mated. Then their crossing sites are randomly chosen as indicated. We assume that there is no mutation.

w_1	w_2	w_3
0	1	1
0	-1	1
-1	0	-1
1	1	-1

Iteration 3. New population

After the above crossover breeding, the new population with the fitness values is:

w_1	w_2	w_3	f	
0	1	1	6	
0	-1	1	4	
-1	1	-1	4	
1	0	-1	8	← Correct solution
			22	

The last solution, $(w_1, w_2, w_3) = (1, 0, -1)$, is a correct one, that is, it matches all y_c's to the corresponding y_t's as follows.

x_1	x_2	x_3	y_c	y_t	$y_c = y_t$?
0	0	0	1	1	yes
0	0	1	0	0	yes
0	1	0	1	1	yes
0	1	1	0	0	yes
1	0	0	1	1	yes
1	0	1	1	1	yes
1	1	0	1	1	yes
1	1	1	1	1	yes
					8 yes

Additional remarks

Using the simple example of this section, we have learned how genetic algorithms can be applied to machine learning problems. The specific expression employed in this section, a weighted sum, is just one example of representing input-to-output mapping . Also, the input-to-output mapping is a very special case of a much broader concept of knowledge representation. Other common forms of knowledge representation include rule-based, frame-based, and computer programs. Genetic algorithms can be applied to these other forms of knowledge representation. Genetic algorithms applied to generate computer programs are briefly discussed in Section 4.7, and applied to rule-based systems in Section 4.8.

The specific form of a weighted sum is the most commonly employed expression in neural networks discussed in Chapters 2 and 3. Genetic algorithms can be applied to neural networks, that is, these two paradigms are used as hybrid systems. For example, in many practical applications, the most difficult part of the backpropagation model discussed in Chapter 2 is training of neural networks. It may take hours, days, or even weeks, before a neural network is trained using the fastest computer, or even worse, it may never converge. Genetic algorithms may help for this problem. The basic idea is as follows:

A set of weights, w_{ij}'s, w'_{ij}'s, and so on, including the thresholds for the network configuration is a solution of the genetic algorithm. This is the same idea as the example of this section. For example, if the network has a total of 100 weights, each genetic algorithm solution consists of 100 numbers instead of three, (w_1, w_2,

w_3). Each weight may be a randomly picked number between -0.5 and 0.5. We generate, say, 20 sets or solutions for a population. Among the 20 solutions, some would be better than the others. The goodness or fitness of the solutions can be measured by the total error between the computed and target patterns; the lesser the error, the better the solution. By applying the genetic algorithm, some bad solutions will disappear. Good parts of good solutions, perhaps representing near-optimal weights associated with particular segments of the neural network may be combined by crossover breeding, producing even better solutions.

We can further extend the above process to include other parameters such as learning and momentum rates in the solutions. Values of these parameters are also assigned randomly, for example, the learning rate between 0.3 and 0.7, and the momentum rate between 0.8 and 1.0, etc. We may perform several backpropagation iterations on the solutions. This time, the fitness may be measured not only by the static total error at a snapshot of a solution, but also the dynamic aspect of the solution, that is, how the solution improves over backpropagation iterations when the weights are adjusted. In this way, in addition to the weights, a good combination of parameter values may be found, and these values can be changed over time.

One major problem of this hybrid approach is that it is computationally very expensive, for both time and storage. We can understand this because even only one solution, neural network computation is often expensive. When we deal with, say, 20 solutions instead of one, computation would cost much more. When training of a network is really hard, however, this hybrid approach is one option we can try.

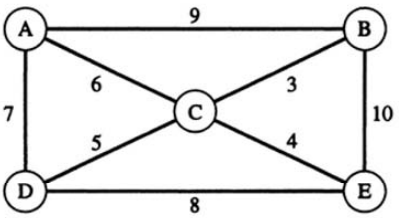
4.5 A Hard Optimization Example: the Traveling Salesman Problem (TSP)

In this section, we will discuss another common form of genetic algorithm application, which involves permutations as solutions. Through this example, we will learn a different way of representing solutions. Subsequent operations require modifications because of the different representation of solutions. This example suggests that we should be flexible for solution representation in general, and devise appropriate operations to fit the representation.

As discussed in the previous chapter, as an application of the Hopfield-Tank neural network model, the TSP is a well-known hard optimization problem. For this reason, the TSP has been chosen as a popular bench mark for many new techniques in solving hard problems. The genetic algorithm described here is not a particularly powerful tool for the TSP, but it is given here to illustrate representing solutions by permutations and the subsequent algorithm. For the convenience of the reader, the problem is stated again in the following.

Problem description

Given an undirected weighted graph, find a shortest tour (a shortest path in which every vertex is visited exactly once, except that the initial and terminal vertices are the same). Fig. 4.3 shows an example of such a graph and its optimal solution. *A*, *B*, etc., are cities and the numbers associated with the edges are the distances between the cities.



Optimal solution:

A —9— B —3— C —4— E —8— D —7— (A)

Fig. 4.3.An example of the traveling salesman problem and its optimal solution.

A genetic algorithm approach to the TSP

Representation of a solution

It is natural to represent each solution as a permutation of the cities, *A*, *B*, etc. The terminal city can be omitted in the representation since it should always be the same as the initial city. For the computation of the total distance of each tour, the terminal city must be counted. Without loss of generality, we can arbitrarily choose *A* as an initial city. (Cyclically permuted solutions belong to the same solution class. For example, in a graph of four cities, *A*, *B*, *C*, and *D*, solutions *ABCD*, *BCDA*, *CDAB*, and *DABC* all represent equivalent solutions. Also note that for each solution, there is another solution by traveling the cities in backward; e.g., *ABCD* and *ADCB*.)

By representing each solution by a permutation of the cities, each city will be visited exactly once. Not every permutation, however, represents a valid solution, since some cities are not directly connected (for example, *A* and *E* in Fig. 4.3). One practical approach is to assign an artificially large distance (e.g., 100) between cities that are not directly connected. In this way, this type of invalid solution that contains consecutive non-adjacent cities will disappear in one iteration of the reproduction process.

Fitness function

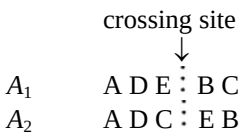
Our objective here is to minimize the total distance of each tour. Hence, it is natural to choose a fitness function to reflect this objective, i.e., the lesser the total

distance, s, the more the fitness function value, f. Generally speaking, there are many such functions. For example, we can have $f = 1/s$, $f = 1/s^2$, or $f = k - s$, where k is a positive constant that makes $f \geq 0$.

The selection of the fitness function is up to us. The selection affects the performance of the genetic algorithm, e.g., the total number of iterations required to find an optimal solution or, more importantly, whether we will be able to find an optimal solution. There is no general formula to design the best fitness function. When we do not adequately reflect the goodness of the solution in the fitness function, finding an optimal solution will not be effective. When we reflect the goodness of the solution to the fitness too weakly, longer iterations will be required. On the other hand, if we are too eager to be effective in searching for an optimal solution and over-emphasize the goodness of the solution, then we might end up with too many clones within a few generations (iterations). Such loss of diversity in the population is undesirable; the best solution obtained from such a population may be far from optimal.

Crossover operations

We realize that an ordinary crossover operation that randomly picks out a crossing site and then swaps string parts does not generally yield valid solutions. For example, in Fig. 4.3, suppose we have the following two solutions for a crossover operation:



The resulting two new strings, A'_1 and A'_2, then, will be:

A'_1	A D E E B
A'_2	A D C B C

in which the same cities are visited more than once. Generally, because of operations such as crossover, which are unique to genetic algorithms, special techniques need to be devised for the representations of solutions. For our TSP, we index the cities and manipulate the indices.

Re-representation of a solution by an index list

Make an *initial index table* by indexing numbers 0, 1, 2, ..., to Cities *A*, *B*, *C*, ... For example, if the number of cities is five, then we have the following:

City	A	B	C	D	E
Index	0	1	2	3	4

Initialization of an index list. Pick out the index for the first city to be visited and enter it in an empty *index list*. (When we arbitrarily choose *A* to be the first city, the

first index will always be 0.)

Repeat the following for all the remaining cities:

> Remove the city from the index table and re-index the remaining cities.
> Pick out the index for the next city to be visited and add it to the index list.

The resulting index list represents the tour (solution).

Example. A D E B C

Index Table					Next City and Index	Index List
A	B	C	D	E	A	
0	1	2	3	4	0	0
B	C	D	E		D	
0	1	2	3		2	02
B	C	E			E	
0	1	2			2	022
B	C				B	
0	1				0	0220
C					C	
0					0	02200 ←

The last index list, 02200, represents a solution of A D E B C.

Crossover operations on index lists

Suppose that we apply this index list method to two solutions, A D E B C and A D C E B. The first solution can be represented as 02200 as we saw in the previous example, and the second solution, A D C E B, can be represented similarly as 02110. Let us perform an ordinary crossover operation on these index lists assuming the following crossing site:

		crossing site ↓
A_1	*A D E B C*	0 2 2 ⋮ 0 0
A_2	*A D C E B*	0 2 1 ⋮ 1 0

The resulting new solutions are:

		New solution
A'_1	0 2 2 ⋮ 1 0	*A D E C B*
A'_2	0 2 1 ⋮ 0 0	*A D C B E*

To obtain a solution of cities, *A D E C B*, back from an index list, 0 2 2 1 0, we can use a work sheet with the same format as above to find the index list from the city list. The difference is in finding the cities from the indices.

Index Table					Next City and Index	Index List
A	B	C	D	E	A	
0	1	2	3	4	0	02210
B	C	D	E		D	
0	1	2	3		2	2210
B	C	E				
0	1	2				

In the above, A'_1, *A D E C B*, with a total distance of 31, happens to be an optimal solution. The second solution A'_2, *A D C B E*, is invalid since *A* and *E* are not directly connected.

Are the solutions obtained by crossing over index lists guaranteed to yield permutations of cities? Yes, they are. In general, for *n* cities, each index list has *n* indices. If we can start from any city, the first index ranges from 0 to *n* - 1 (if we always start from *A*, then the first index is always 0). The second index ranges from 0 to *n* - 2, ..., and the last index is always 0. When we swap parts of index lists, each index in the lists still satisfies its range in the position. Hence, cities can be picked out from index tables and these cities are unique in each solution.

Mutation

Mutation represents unpredictable small changes in the solutions. For example, occasional random swapping of two cities can be treated as mutation.

Global procedure and iteration termination conditions

There are many variations on how to carry out the algorithm. The following is a simple one. For a given weighted graph, we set a population size. The first generations (iteration) of solutions in the population are randomly generated. Then iterations proceed as described above. In each generation, the best (shortest) total distance among the solutions may be recorded to keep track of the progress over the iterations. When the best total distance does not improve for a certain number (say, 5 or 10) of consecutive iterations, we may assume that the solutions are converged and the best solution for this particular run is reached. As a safeguard, we may set the maximum number of iterations to a certain fixed constant (e.g., 100,

500, or 1000, depending on the size of the problem) so that the algorithm will not continue for an excessive amount of time.

There is no theoretical guarantee that the "optimal" solutions from genetic algorithms are truly optimal. That is, solutions can converge before reaching the true optimal; this phenomena is called **premature convergence**. For this reason, a certain number of (say, 10) runs may be carried out; each time we start with a different initial population by using a different random number seed. The best solutions from different runs can be compared, and the best among the best solutions can be chosen as "optimal" from the experiment. For practical purposes, with careful implementation of the technique, this process should provide adequate solutions for most problems.

Partially matched crossover (PMX) operation

When dealing with permutations as solutions, simple crossover operations will result in invalid solutions. In the above TSP, for example, a crossover of two solutions, A D E ⋮ B C and A D C ⋮ E B, yields two invalid solutions, A D E E B and A D C B C. To avoid this problem, we introduced the method using index lists. The PMX operation is another technique which directly operates on permutations and still gives permutations. It can be used not only for the TSP, but also any other problems that involve permutations.

We illustrate this technique by an example. We will use numbers 1, 2, ..., instead of letters A, B, ... Assume that each solution in this example is a permutation of 10 numbers, 1, 2, ..., 10. Pick two crossing sites uniformly at random. The substrings between the sites are called the matching sections.

A_1 9 8 4 ⋮ 5 6 7 ⋮ 1 3 2 10
A_2 8 7 1 ⋮ 2 3 10 ⋮ 9 5 4 6

Consider the two-element permutations in the matching sections: (5, 2), (6, 3), and (7, 10). Permute each of these two-element permutations in each string of A_1 and A_2. For example, when the two-element permutation (5, 2) is applied to A_1: 9 8 4 ⋮ 5 6 7 ⋮ 1 3 2 10, we will have 9 8 4 ⋮ 2 6 7 ⋮ 1 3 5 10. When the next two-element permutation (6, 3) is applied to 9 8 4 ⋮ 2 6 7 ⋮ 1 3 5 10, we will have 9 8 4 ⋮ 2 3 7 ⋮ 1 6 5 10. Finally when the permutation (7, 10) is applied, we will have 9 8 4 ⋮ 2 3 10 ⋮ 1 6 5 7. String A_2 is permuted similarly. The resulting two new strings, A'_1 and A'_2, then will be:

A'_1 9 8 4 ⋮ 2 3 10 ⋮ 1 6 5 7
A'_2 8 10 1 ⋮ 5 6 7 ⋮ 9 2 4 3

The numbers in the matching sections of A_1 and A_2, (5, 6, 7) and (2, 3, 10), have been in effect swapped between the two solutions as if in an ordinary crossover operation. The two new strings, A'_1 and A'_2, remain as valid permutations, since the numbers are actually permuted within each string.

4.6 Schemata

In the preceding sections we have learned the basic concept and steps of genetic algorithms. They are like a cookbook recipe; to randomly generate an initial population, followed by processes such as reproduction, crossover breeding and mutation. In this section, we will discuss some theoretical aspects of genetic algorithms. For example, we may wonder how a good gene or a part of a solution is likely to survive better than a worse one. What is the probability of the survival rate or the expected count of such a gene? The answer will be derived as the schema theorem or the fundamental theorem of genetic algorithms in this section.

A **schema** (or **similarity template**) is a pattern of solutions. For example, suppose that a solution is a string of four bits. Given two solutions, 1010 and 1011, we may say the common pattern of the two strings is that the first three bits are 101. This statement, "the first three bits are 101," can be expressed more compactly by introducing a new wild card character "*", which stands for "*don't care*" or "either 0 or 1" in general. Note that we are extending our alphabet from {0,1} to {0, 1, *} to represent the schemata. The two solutions 1010 and 1011 then correspond to schema 101*.

Counting problems of schemata

We will consider some counting problems of schemata and solutions, which are helpful to analyze characteristics related to schemata. In the above, we saw the two solutions 1010 and 1011 correspond to schema 101*. Conversely, a schema corresponds to solutions. For example, schema **11 can correspond to four solutions: 0011, 0111, 1011, and 1111. In general, a schema containing l "*"s can correspond to 2^l solutions since each "*" can be either 0 or 1.

Conversely again, a solution can correspond to many schemata. For example, 1010 can correspond to 101*, ****, 1010, etc. There are 2^4 schemata for 1010 since each bit can be either its actual value or *. In general, a particular solution of length L contains 2^L schemata. For $L = 2$ and a solution of 00, for example, there are four schemata: 00, 0*, *0, and **.

Let us consider a population of size n, where each solution has length L. The lower bound of the number of schemata contained in this population is 2^L, and the upper bound is $n \cdot 2^L$. The lower bound occurs when all n solutions are the same (degenerate). In this case, since all the solutions are the same, the number of schemata reduces to the number for a single string, which is discussed above. For example, for $L = 2$ and $n = 2$, a population of two degenerate solutions, 00 and 00, contains 4 schemata: 00, 0*, *0, and **. To consider the upper bound, assume that all the solutions are different. Since each of n solutions can contain 2^L schemata, the upper bound is $n \cdot 2^L$. The actual number, however, is usually smaller since different solutions can contain the same schemata; e.g., schema **···* of length L is contained in all n solutions. For example, for $L = 2$ and $n = 2$, a population of two solutions, 00 and 11, contains 7 schemata: 00, 0*, *0, **, 11, 1*, and *1. Note that 7 is smaller than $n \cdot 2^L = 8$, since schema ** is contained in both 00 and 11.

The next question is how many different schemata are possible in L-bit strings.

The answer is 3^L since each bit can take one of three possible values; 0, 1, or *. For example, for 2-bit strings, there are $3^2 = 9$ schemata: 00, 01, 0*, 10, 11, 1*, *0, *1, and **.

We represent strings with capital letters and individual characters within a string with lower case letters subscripted by their position, e.g., A = 1010 may be symbolically represented by

$$A = a_1\ a_2\ a_3\ a_4$$

where $a_1 = 1$ for position 1, etc. We sometimes call a_i a **gene** and a value 0 or 1 an **allele**. We count crossing sites 1, 2, ..., starting from the left-most place, i.e., between a_1 and a_2:

$$\begin{array}{lcccccc} \text{Crossing site} & & 1 & & 2 & & 3 \\ A = & a_1 & \vdots & a_2 & \vdots & a_3 & \vdots\ a_4 \end{array}$$

A population of n strings can be represented by **A**, where **A** = (A_j, j = 1, 2, ..., n). (**A** is like a vector and A_j is its component.) In particular, to represent a population at time (generation, or iteration) t, we can write **A**(t).

The **order** of a schema H, denoted as $o(H)$, is the number of fixed positions (i.e., 0's and 1's), or equivalently, (the length of schema, L) - (the number of '*'s). The **defining length** of a schema H, denoted as $\delta(H)$, is the distance between the first and last fixed string positions. For example, o(01*1) = 3, δ(01*1) = 4 - 1 = 3, o(1***) = 1, δ (1***) = 1 - 1 = 0. The order and defining length will be used in the next subsection.

4.6.1 Changes of Schemata Over Generations

In this subsection, we will determine how the number of strings representing a particular schema H changes over iteration time t. We will evaluate the changes in each of three steps; reproduction, crossover, and mutation. We will then combine the changes into one formula to reflect all three processes.

Number of changes of strings of a schema as a result of reproduction

Let m be the number of strings of a particular schema H contained within the population **A**(t). We write $m = m(H, t)$. For example, suppose that **A**(t) is a population of four four-bit strings: 1010, 1011, 0000, and 0001. Schema H = 10** is contained in **A**(t), and $m(H, t) = 2$, for the two strings 1010 and 1011.

During reproduction, a string, A_i is copied according to its fitness, i.e., the probability $p_i = f_i / \Sigma f_i$. The expected number (count) of strings of A_i in the new mating pool is np_i. Assume that $f(H)$ is the average fitness of the strings representing schema H at time t. Then

$$m(H, t+1) = m(H, t) \cdot n \cdot \frac{f(H)}{\sum f_i}$$

We may understand this equation as follows: $f(H)/\Sigma f_i$ is the probability of one string with the average fitness for schema H. This probability times $m(H, t)$ gives the total probability for the group of strings representing schema H. Finally, multiplying this quantity by n, the population size, determines the expected number of the generated strings representing schema H at $t + 1$. Using the average fitness of the entire population, $f_{av} = (\Sigma f_i)/n$, the above can be rewritten as

$$m(H, t + 1) = m(H, t) \cdot \frac{f(H)}{f_{av}}.$$

That is, a particular schema grows (or decays) as the ratio of the average fitness of the schema to the average fitness of the population. For example, if $f_{av} = 100$ and $f(H)$ for a particular schema is 120 at time t, then $m(H, t + 1)$ will be 1.2 times $m(H, t)$. We note that values of both f_{av} and $f(H)$ change over time t. For the moment, as a gross approximation, if we assume this ratio $f(H)/f_{av} = 1.2$ to be stationary, i.e., it does not change over time starting at $t = 0$, then $m(H,t) = m(H, 0) \cdot (1.2)^t$. More generally, if the ratio is α, rather than 1.2, then $m(H, t) = m(H, 0) \cdot \alpha^t$. That is, $m(H, t)$ is an exponential function of time, t; the function grows when $\alpha > 1$, and decays when $\alpha < 1$. Exponential functions grow or decay much faster than other common functions such as polynomials (e.g., t,t^2, etc.).

Effect of crossover on schemata

The effect by the pattern of the schema and the crossing site

Example:

crossing site ↓

A = 01 ⋮ 0100
H_1 = *1 ⋮ ***0
H_2 = ** ⋮ *10*

In the above example, both schemata H_1 and H_2 are contained in string A. We want to see whether H_1 and H_2 will be still contained in the offspring of A. When A is mated with a different string, the characteristics of the schema H_1 may be lost since the two bits (1 in position 2 and 0 in position 6) will be placed in different offspring. The characteristics of schema H_2 (i.e., 1 in position 4 and 0 in position 5), however, will definitely remain, since these two positions go together to a single offspring.

As we see in this example, whether the characteristics of the schema survive or are destroyed depends on two major factors. One is the pattern of the schema; the other is the crossing site. For example, H_1 above will survive only if the crossing site is 1, whereas H_2 will survive most of time except the case where the site is 4. For H_1, *1***0, for example, there are $L - 1 = 6 - 1 = 5$ possible crossing sites:

L - 1 possible crossing sites
for a schema of length L

$$H_1 = *\,\vdots\,1\,\vdots\,*\,\vdots\,*\,\vdots\,*\,\vdots\,0$$

Among those, if the site falls on one of the positions within the defining length (i.e., the last fixed position 6 - the first fixed position 2) 4, then H_1 will be destroyed. That is, the probability that the characteristics of H_1 will be destroyed in the offspring is 4/5. Here we assumed that all the possible crossing sites are equally likely to be chosen. In general, the probability that a schema will be destroyed is

$$p_d = \frac{\delta(H)}{L-1}.$$

The effect by mate

In addition to the pattern of the schema and the crossing site, there is another factor that affects whether the characteristics of a schema survive or not: the pattern of the mate, i.e., the second parent of the offspring. When the pattern of the mate contains parts of the schema, the characteristics of the schema may survive. (In the above example of H_1 = *1 $\vdots$ ***0, the mate has either *1 or ***0, or both.) Consider the following:

Example. A pattern of the mate affects the survival of the schema characteristics.

crossing site

$$H_1 = \quad *1* \;\vdots\; ***0$$
$$A = \quad 011 \;\vdots\; 1000$$

Since the crossing site is inside of the schema, the characteristics of the schema would be destroyed using the first two major factors. The actual offspring, however, depend on the mate. For example, if the mate B is:

$$B_1 = \quad 000 \;\vdots\; 1111$$

Then the offspring are:

$$A' = 011 \;\vdots\; 1111$$
$$B'_1 = 000 \;\vdots\; 1000$$

and the characteristics are lost. However, if the mate B is:

$$B_2 = \quad 000 \;\vdots\; 0000$$

Then the offspring are:

$$A' = 011 \vdots 0000$$
$$B'_2 = 000 \vdots 1000$$

and the characteristics survive in A'.

The combined effect

In the following, we assume that the survival of the schema characteristics by mate is not a dominant factor, and include its effect in the form of inequalities. When the string length L is large, and each population has diversified solutions, this effect by mate will be small. The probability that a schema will be destroyed is now:

$$p_d \leq \frac{\delta(H)}{L-1}$$

Note that the equality "=" in the previous equation for p_d is replaced with the inequality "≤", since there can be some additional survivals from mates. In the following, inequalities, "≤" and "≥" are due to the effect by mate. The probability p_s that a schema will survive is 1 - p_d, i.e.,

$$p_s = 1 - p_d \geq 1 - \frac{\delta(H)}{L-1}.$$

Let the crossover process itself be performed with a certain probability, p_c (i.e., the crossover will not take place with the probability 1 - p_c). Then the probability that a schemata will be destroyed is

$$p_d \leq p_c \cdot \frac{\delta(H)}{L-1}.$$

The probability of survival in this case is

$$p_s = 1 - p_d \geq 1 - p_c \cdot \frac{\delta(H)}{L-1}.$$

Combined effect of reproduction and crossover

As discussed before, after reproduction, a particular schema grows (or decays) by:

$$m(H, t+1) = m(H, t) \cdot \frac{f(H)}{f_{av}}$$

Assuming the reproduction and crossover operations are mutually independent, the growth (or decay) of a particular schema after these two operations is the product of the two expressions corresponding to the two operations:

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{f_{av}} \left[1 - p_c \frac{\delta(H)}{L-1}\right].$$

Again, assuming that the factor stays constant, schema H grows exponentially with the factor. The factor depends on two parameters: $f(H)$, the fitness, and $\delta(H)$, the defining length. Those schemata with high fitness values and short defining lengths are more likely to survive and grow.

Mutation effect

Let a single position in string A be changed by mutation with probability p_m (i.e., a single position does not change with probability $1 - p_m$). In order for a schema H contained in A to survive, all the fixed positions of H must stay the same in A. For example, suppose $H = 1 * * 0$, and because of mutation, $A = 1110$ is changed to $A' = 0110$; then H is the lost in A'. The survival probability of each position is $1 - p_m$, and there are $o(H)$ fixed positions (remember $o(H)$, the order of H, is the number of fixed positions in H). Hence, the survival probability of the schema H is obtained by multiplying $1 - p_m$ by itself $o(H)$ times; i.e., the survival probability = $(1 - p_m)^{o(H)}$. For small values of p_m (i.e., $p_m << 1$), we have $(1 - p_m)^{o(H)} \approx 1 - o(H) \cdot p_m$ (by the Taylor series expansion of calculus). Summarizing all the above, and using the following approximation:

$$\left[1 - p_c \frac{\delta(H)}{L-1}\right] \cdot (1 - o(H) \cdot p_m) \approx 1 - p_c \cdot \frac{\delta(H)}{L-1} - o(H) \cdot p_m$$

we have

$$m(H, t+1) \geq m(H, t) \cdot \frac{f(H)}{f_{av}} \left[1 - p_c \frac{\delta(H)}{L-1} - o(H).p_m\right]$$

This is the **schema theorem** or the **fundamental theorem of genetic algorithms**. This determines the expected count of strings of a particular schema with combined effects of reproduction, crossover, and mutation.

4.6.2 Example of Schema Processing

In this example, each population is a set of four four-bit strings. The fitness value of each string is simply the binary number represented by the string as a whole, as appeared in Section 4.3. The following table shows a randomly generated initial population, the associated values of the strings, and actual counts of newly generated strings. We assume that $p_c = 1$, i.e., the crossover process takes place in all strings.

String processing 1

i	Randomly Generated Initial Population A_i	Fitness f_i	Probability p_i	Expected Count $n \cdot p_i$	Actual Count in String Processing 2
1	1010	10	0.345	1.379	1
2	0010	2	0.069	0.276	0
3	0101	5	0.172	0.690	1
4	1100	12	0.414	1.655	2
	Total	29	1.000	4.000	4
	Average	7.25	0.250	1.000	1
	Max	12	0.414	1.655	2

In the following, we will consider two schema, $H_1 = 0***$ and $H_2 = 1*0*$.

Schemata processing

		i for Representative A_i	Count	f_i	Schema Average Fitness, $f(H)$
H_1	0***	2,3	2	2, 5	3.5
H_2	1*0*	4	1	12	12

String processing 2-1, reproduction

Suppose that the following four new strings A_i are reproduced. Then mates and crossing sites are selected, all at random, as shown.

i	A_i	i for Mate
1	110 ⋮ 0	2
2	010 ⋮ 1	1
3	10 ⋮ 10	4
4	11 ⋮ 00	3

Schema processing 2-1, after reproduction

		i for Representative A_i	Expected Count	Actual Count
H_1	0***	2	0.966	1
H_2	1*0*	1, 4	1.655	2

The expected counts are determined by the formula $m(H, t + 1) = m(\mathrm{H}, t) \cdot f(H)/f_{av}$. For H_1, we have $m(H_1, 2) = m(H_1, 1) \cdot f(H_1)/f_{av} = 2 \times 3.5/7.25 = 0.966$. For H_2, we have $m(H_2, 2) = m(H_2, 1) \cdot f(H_2)/f_{av} = 1 \times 12/7.25 = 1.655$.

String processing 2-2, crossover breeding

Based on the outcome of String processing 2a, the following new population is generated:

i	A_i	f_i
1	1101	13
2	0100	4
3	1000	8
4	1110	14
Total		39
Average		9.75
Max		14

Schema processing 2-2, after crossover breeding

		i for Representative A_i'	Expected Count	Actual Count
H_1	0***	2	0.966	1
H_2	1*0*	1, 3	0.552	2

The expected counts are determined by the formula $m(H, t+1) = m(H, t) \cdot \{f(H)/f_{av}\}$ $[1 - p_c \cdot \delta(H)/(L - 1)]$. For H_1, we have $m(H_1, 2) = 0.966 \times [1 - 1 \times 0/3] = 0.966$. For H_2, we have $m(H_2, 2) = 1.655 \times [1 - 1 \times 2/3] = 0.552$.

The significance of schemata

As we have seen in this section, the concept of schemata leads to some theoretical foundations of genetic algorithms, represented by the schema theorem. The schema theorem tells us which and how parts of the solutions are likely to survive and grow as iterations proceed. In effect, the genetic algorithm searches for good partial solutions. Such underlying mechanism would be more effective than working on the entire solution string which has a large number of combinatorial alternatives. A partial good solution can then become a *building block*. Building blocks for different portions of a string are found, then they are combined to form a good solution for the entire string. If we can find such building blocks by inspection, they could also be manually placed together to form good solutions. The concept is a widely known approach called the *divide-and-conquer* method. This is the basic idea of the schemata.

There are, however, some questions raised whether the above view on schemata, the good ones always survive and grow, is true. For one, no automatic selection, calculation or operation is ever explicitly and directly performed on specific schema by the genetic algorithm. In fact, there have been papers showing examples in which this view is not necessarily true. For example, suppose that the schema 11*** has a high average fitness. After a certain number of generations, almost all

solutions may be in form of 11***. Then the schema **00* for these populations with heavy concentrations on 11*** may not be given an accurate evaluation, since in effect, we are dealing with the schema 1100*, rather than intended **00*.

Theoretical analysis of how actually the schema theorem works is a difficult problem. For most practical applications, however, it basically appears to hold. For more, see Mitchell (1996) as a starting point.

4.7 Genetic Programming

Genetic programming is a subfield of genetic algorithms, where each solution is a computer program. To understand the basic idea, we present a simple example below.

Example. Boolean exclusive-or (XOR) function

The goal of the problem is to express the boolean XOR function by using the three basic boolean operators, OR, AND, and NOT. The OR, AND, and XOR are binary operators (which means that each operator takes two operands), and NOT is a unary operator (which means it takes one operand). In this example, the binary operators are used as prefix operators, e.g., an expression can be (AND *P Q*), rather than (*P* AND *Q*) for infix operators. Here *P* and *Q* are boolean variables, i.e., each of *P* and *Q* takes one of two values 0 (False) and 1 (True).

Each solution for this problem is a boolean expression consisting of boolean operands and the OR, AND, and NOT operators. Although a solution of a boolean expression is not a typical program, we can extend this format to include common program elements such as assignment, loop, and if statements. Manipulation of these target programs can be implemented in any common computer language, but the most convenient ones are symbolic AI languages such as Lisp and Prolog.

The XOR function takes two operands, *P* and *Q*. Since each of *P* and *Q* can take one of two values, 0 and 1, there are four possible values for *P* and *Q*. The XOR function is defined as follows for these four cases:

P	*Q*	(XOR *P Q*)
0	0	0
0	1	1
1	0	1
1	1	0

Our task is to generate this XOR function by using OR, AND, and NOT. As in an ordinary genetic algorithm, we will randomly generate the initial population of solutions. Each solution is a boolean expression involving OR, AND, and NOT operators and the operands *P* and *Q*. Fitness values are determined, crossover breeding is carried out, and occasional mutation may take place. For example, we may have the following two solutions at some point:

A_1	(AND(OR P Q)	⋮	(AND P Q))
A_2	(AND(OR (NOT P) Q))	⋮	(NOT (AND P Q)))

Each solution can be interpreted as representing a tree. Each operator is the root of the tree or a subtree; each operand or a compound operand (a subtree) is a node of the tree. A crossing site may be picked out randomly, and *independently for each solution*, at the root of a subtree. Randomly chosen crossing sites for our particular example are indicated by ⋮ above. By crossover operation, we swap the entire subtrees of the two solutions. In our example, the offspring will be:

A'_1	(AND (OR P Q)	⋮	(NOT (AND P Q)))
A'_2	(AND (OR (NOT P) Q))	⋮	(AND P Q))

The fitness function for this problem may be defined as the number of P and Q combinations that matches the answer, XOR. A solution with the fitness value of 4 will be an optimal solution. The boolean values for different P and Q values and the fitness values for the four solutions, A_1, A_2, A'_1, and A'_2, are summarized as follows:

P	Q	(XOR P Q)	A_1	A_2	A'	A'_2
0	0	0	0	1	0	0
0	1	1	0	1	1	0
1	0	1	0	0	1	0
1	1	0	1	0	0	1
		f =	1	2	4	1

We can see that A'_1 is an optimal solution.

In the previous example, we sought any boolean expression that realizes (XOR P Q). Since such a boolean expression is not unique, we could have other answers, possibly longer expressions or ones containing redundant sub-expressions. To discourage such longer answers, we can modify the fitness function. For example, we may add new terms: 3 - 0.1 × (number of operators) to the original fitness function.

Mutation in this example can be performed by negating an operand or compound operand. For example, P may be changed to (NOT P), or (NOT (AND P Q)) may be changed to (AND P Q). This is analogous to changing a binary bit from 0 to 1, or from 1 to 0 in an ordinary genetic algorithm.

To extend the solution format of boolean expressions to include common program elements such as assignment, loop, and if, statements, we may represent them in prefix form. For example, the assignment "$x = a * b + c$" can be represented as (= x (+ (* a b) c)); "if $x > 0$, $y = 1$" can be represented by (if (> x 0) (= y 1)). These expressions can be interpreted as representing trees, and crossing sites can be picked out at the roots of subtrees. Fitness functions will be defined to reflect whatever the programs are supposed to achieve.

4.8 Additional Remarks

Classifier systems: applying genetic algorithms to machine learning on rule-based systems

In Section 4.4, we have seen an application of a genetic algorithm to one particular form of machine learning, an input-to-output mapping by a weighted sum. In that example, the machine learns by finding correct values of the weights. Machine learning refers to any technique by which the machine (computer) gets smarter, that is, more knowledgeable, by itself.

A knowledge base can be represented by a set of rules (called production rules). Each rule can be in the form of "if <condition> then <action>." (The name is "production rule" since it produces the action based on the condition.) A rule (or a set of rules) can be a solution of the genetic algorithm. The condition and action parts can be represented as a string or any other appropriate form. A system consisting of such solutions is called a **classifier system**. We can define a fitness function to reflect the goodness of each rule in terms of achieving the goal. We can select mating pools for good rules, then perform crossover breeding to create new offspring rules. Occasional mutation may create unexpectedly innovative ideas as new rules. In these processes, the knowledge base will get increasingly better automatically.

The significance of genetic algorithms

As we have seen, the major ingredients of the genetic algorithm are random generation of solutions, mating (crossover operations) of the solutions, mutation of the solutions, and evolution of the solutions based on the fitness values. In very general terms, the genetic algorithm is a *guided random search method*. We use randomness, which depends on chance, but we also incorporate some guidance to search solutions effectively. Neural network algorithms are also guided random search methods. For example, in the backpropagation model, initial weight values are randomly assigned, but their succeeding values are guided to change in a way to minimize the error between the output and target values. The genetic algorithm is a probabilistic guided random search since it guides its search based on probability. The guided random search is in contrast to blind random search methods. The Monte Carlo method for random number simulation is a typical blind random search method. Examples of blind random searches include random number simulations of neutron behavior in an atomic reactor lead shield wall, and queuing lines at bank tellers.

As we also have seen, genetic algorithms can be used for optimization problems and machine learning. When we think of a search space to find an optimal or target point, the genetic algorithms search from a population of points, rather than a single point in the space. The genetic algorithms use the objective (fitness) function itself, not derivatives or other auxiliary quantities. The characteristics of a genetic algorithm as an optimization technique are common to the guided random search method. For example, the search can be trapped in a local minima. The answers obtained from genetic algorithms are not guaranteed to be globally optimal. This

contrasts with most classical optimization techniques such as the Lagrange multiplier method in calculus, and linear or dynamic programming methods in operations research. These classical methods usually guarantee optimal solutions. Why, then, bother to use genetic algorithms that do not guarantee optimal solutions? The answer is because there are so many hard problems for which the classical methods do not work well or take too much computation time. Guided random search methods, such as genetic algorithms, work for some of these problems. One important practical application category is hybrid systems, i.e., combinations of genetic algorithms and other areas such as neural networks, fuzzy systems, and expert systems.

The advantages and disadvantages of genetic algorithms are somewhat similar to those for neural networks. Some advantages include: self-guidance, self-organization, machine learning capability, robustness, flexibility, simple and straightforward computation, and easy implementation of parallelism. Disadvantages include the chance-dependent outcome and lengthy computation time, yet we may or may not obtain satisfactory solutions.

Many general and theoretical problems of genetic algorithms remain to be further investigated. They include: On what types of problems will the genetic algorithms be effective or not effective. Compared with other methods, such as neural networks, for what types of problems do genetic algorithms outperform others? How can we theoretically determine the best parameter values, such as the population size, mutation rate, etc.? How do the basic operations such as reproduction, crossover breeding, and mutation affect the macroscopic behavior of genetic algorithms such as convergence of solutions.

Further Reading

Well-written books for further study of general genetic algorithms:

D.E. Goldberg, *Genetic Algorithms in Search, Optimization & Machine Learning*, Addison-Wesley, 1989.

M. Mitchell, *An Introduction to Genetic Algorithms*, MIT Press, 1996.

Series, each carrying many books:

The *Complex Adaptive Systems Series* published by MIT Press carries many books including the following two titles that cover some fundamentals of genetic algorithms and extensive materials on genetic programming:

J.R. Koza, *Genetic Programming*, MIT Press, 1992.

J.R. Koza, *Genetic Programming II*, MIT Press, 1994.

Springer publishes the following three series. *Genetic Algorithms and Evolutionary Computation*, *Natural Computing*, and *Studies in Computational Intelligence*. The first series lists the following book.

D.E. Goldberg, *The Design of Innovation*, Springer; 2002.

Journals:

IEEE Transactions on Evolutionary Computation
Evolutionary Computation, MIT Press.
Artificial Life, MIT Press.
Machine Learning, Springer.

Conference proceedings that include many theoretical and practical developments in the field.

Genetic and Evolutionary Computation Conference (*GECCO*).

IEEE Congress on Evolutionary Computation.

The following workshop proceedings have been published every other year since 1991 and focus on theoretical foundations in the field.

Foundations of Genetic Algorithms, Morgan Kaufmann.

Additional references:

J.H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, MIT Press, 1992.

Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd Ed., Springer-Verlag, 1998.

5 Fuzzy Systems

5.1 Introduction

The term "fuzzy systems" here includes fuzzy sets, logic, algorithms, and control. The fundamental idea common to all of these "fuzzy domains" is the exploitation of the concept of fuzziness. The key concept of fuzziness is that it allows a gradual and continuous transition, say, from 0 to 1, rather than a crisp and abrupt change between binary values of 0 and 1. This may sound too simple, but traditional set theory and logic deal with only discrete rather than continuous values. For example, an element in an **ordinary set** (**crisp set**) either belongs to the set (which may be represented by a 1) or does not (which may be represented by a 0). Similarly, in ordinary logic, a proposition is either true (which may be represented by a 1) or false (which may be represented by a 0). We had never dealt with a situation like an element which belongs to a set with a degree of 30% or a proposition is half true. Fuzzy systems extend the traditional fields by incorporating such partial truthfulness. We may say the concept of fuzziness is the science of continuum, especially for traditionally discrete disciplines. As we will see in this chapter, extending crisp to continuous in these fields turns out to be useful for certain types of applications.

In what areas are fuzzy systems effective and why?

Primary types of applications for which fuzzy systems are particularly useful are difficult cases where traditional techniques do not work well. The most successful domain in terms of practicality has been in fuzzy control of various physical or chemical characteristics such as temperature, electric current, flow of liquid/gas, motion of machines, etc. Also, fuzzy systems can be obtained by applying the principles of fuzzy sets and logic to other areas. For example: fuzzy knowledge-based systems, such as fuzzy expert systems, which may use fuzzy if-then rules; "fuzzy software engineering" which may incorporate fuzziness in their programs and data; fuzzy databases which store and retrieve fuzzy information; fuzzy pattern recognition, which deals with fuzzy visual or audio signals; applications to medicine, economics, and management problems which involve fuzzy information processing.

Table. List of selected symbols in this chapter

Symbol	Page	Meaning
$m_A(x)$	126	membership function of fuzzy set A for every $x \in U$, the universe.
A'	126	complement of fuzzy set A.
$\cup_A$	128	taking the unions over all the elements in A.
Σ	128	taking the unions over a finite set.
$\int_U$	128	taking the unions over a set, where the elements in the set take continuous values.
$\int dx$	128	ordinary calculus integration; note "dx" at the end.
$A \times B$	130	cartesian product.
$\cup_{A \times B}$	133	taking the unions over all the elements in the cartesian product $A \times B$.
R o S	135	the composition of two ordinary or fuzzy relations R and S.
$\max_A[x]$	135	taking the maximum of the x values over all the elements of set A.
A=>B	139	"A implies B" or "if A then B"
$\wedge$	149	minimum operator
$\vee$	149	maximum operator

What characteristics of fuzzy systems provide better performance for certain applications? Fuzzy systems are suitable for uncertain or approximate reasoning, especially for the system whose mathematical model is hard to derive. For example, the input and parameter values of a system may involve fuzziness or be inaccurate or incomplete. Similarly, the formulas or inference rules to derive conclusions may be incomplete or inaccurate. Fuzzy logic allows decision making with estimated values under incomplete information. Note that the decision may not be correct, and it can be changed at a later time when additional information is available. Complete lack of information will not support any decision making using any form of logic. For hard problems, conventional nonfuzzy methods are usually expensive and depend on mathematical approximations (e.g. linearization of non-linear problems), which may lead to poor performance. Under such circumstance, fuzzy systems often outperform conventional methods such as a proportional, integral, and differential (PID) control.

Fuzzy system approaches also allow us to represent descriptive or qualitative expressions such as "slow" or "moderately fast" and are easily incorporated with symbolic statements. These expressions and representations are more natural than mathematical equations for many human judgmental rules and statements.

When fuzzy systems are applied to proper problems, particularly the type of problems described above, their responses are typically faster and smoother than with conventional systems. This translates to efficient and more comfortable operations for such tasks as controlling temperature, cruising speed, and so forth. Furthermore, this will save energy, reduce maintenance cost, and prolong machine life. In fuzzy systems, describing the control rules is usually simpler and easier, requiring fewer rules, and thus these systems execute faster than conventional

systems. Fuzzy systems often achieve tractability, robustness, and overall low cost. In turn, all of these factors contribute to better performance. In short, conventional methods are good for simpler problems, while fuzzy systems are suitable for complex problems or applications that involve human descriptive or intuitive thinking.

5.2 Fundamentals of Fuzzy Sets

5.2.1 What is a Fuzzy Set?

In an ordinary (nonfuzzy) set, an element of the universe either belongs to or does not belong to the set. That is, the membership of an element is crisp -- it is either yes or no. A **fuzzy set** is a generalization of an ordinary set by allowing a **degree** (or **grade**) **of membership** for each element. A membership degree is a real number on [0, 1]. In extreme cases, if the degree is 0 the element does not belong to the set, and if 1 the element belongs 100% to the set.

For easy understanding of a fuzzy set, let the people in an organization be the universe. A subset of all the men in this organization is an ordinary, crisp set. Now let us consider a set of "young" people. Obviously the "youngness" is not a step function from 1 to 0 at a certain age, say, 30. It would be natural to associate a degree of youngness to each element, as for example, {Ann/0.8, Bob/0.1, Cathy/1}. Perhaps Ann is 28 years old, Bob 40, and Cathy 23. As we can see in this example, each *element* in a fuzzy set is represented in the format of <*element*>/<*degree*>. (In some books, the order is reversed, i.e., each element is represented in the format of <*degree*>/<*element*>.)

The **membership function** of a set maps each element to its degree. Having every element in a set be associated with a degree of membership, as in the above example, is indeed the foundation of fuzzy sets as well as fuzzy systems. That is, everything in the world is not always a matter of black or white, true or false, yes or no, 1 or 0; it often involves gray areas. Starting with this, we can define or derive various properties and operations. Many of them have counterparts in ordinary sets, relations, and logic, while some are unique to fuzzy systems, as we will see in the following.

Note that the membership degree ($0 \le$ degree ≤ 1; hereafter, sometimes we may simply call it the degree) represents *plausibility* rather than the probability. Because both fuzzy set theory and probability theory deal with values between 0 and 1, sometimes a question is raised how these two theories differ. This book is not intended to cover various aspects of this issue extensively, but a short discussion is provided in Section 5.7. Starting with the basic concept of associating a degree to each element, we can define or derive various relations, operations and properties on fuzzy sets. The following is a simple example of fuzzy sets in form of expressions common to ordinary as well as fuzzy set theories.

Example.

Universe U = set of five specific men: Alan, Bob, Cong, Dave, Eric.
$=\{a, b, c, d, e\}$
Fuzzy set A $=\{x \mid x \text{ is a young man}\}$

Fuzzy set B $=\{x \mid x \text{ is a tall man}\}$

For example, fuzzy set A can be {*a*/0.9, *b*/0, *c*/1, *d*/0, *e*/0.2}, where 0.9 is the membership degree of element a, Alan, and so on. When the membership degree of an element is 0, that element can be omitted totally. Using this convention, A can also be represented as {*a*/0.9, *c*/1, *e*/0.2}.

Characteristics of the elements

In the above example, the elements in *U* are *discrete*, and the *cardinality* of *U*, denoted $|U|$, is *finite*. As in ordinary set theory, variations are possible on these characteristics. For example, the elements are discrete but $|U|$ is infinite, e.g., *U* is the set of natural numbers, 0, 1, 2, ... Or, the elements may be continuous, making $|U|$ is infinite, e.g., $\{x \mid x \text{ is a real number and } -1 \le x \le 1\}$. In all the cases, the degrees is between 0 and 1 inclusive, i.e., $0 \le \text{degree} \le 1$.

Membership function

A membership function gives the degree of membership for every element under consideration.

Example. Membership functions $m(x)$ to be "young" as a function of age x

An example in *table form - discrete*

x	$m(x)$
25	1.0
30	0.5
40	0.1
50	0.0

Another example in *equation form - continuous* (A graph is shown in Fig. 5.1)

$$m(x) = \begin{cases} 1.0 & \text{for } 0 < x \le 25 \\ \dfrac{1}{1+\left\{\left(\dfrac{x-25}{5}\right)^2\right\}} & \text{for } x > 25 \end{cases}$$

For example, the above continuous membership function $m(x)$ can represent a fuzzy set in the universe of all positive real numbers as: $A = \{x/m(x) \mid x \text{ is a positive real number}\}$. The value of $m(x)$ for a specific value of x, e.g., $m(30) = 0.5$, is called the **membership value**.

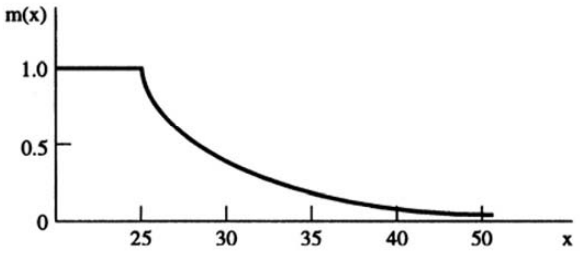


Fig. 5.1.Membership function for youngness.

5.2.2 Basic Fuzzy Set Relations

As in the case of ordinary sets, we can define basic relations on fuzzy sets as follows.

Equality: $A = B$ if and only if $m_A(x) = m_B(x)$ for $x \in U$. Here $m_A(x)$ and $m_B(x)$ are membership functions for fuzzy sets A and B, respectively. "for $x \in U$" means "for every $x \in U$".

Subset: $A \subseteq B$ if and only if $m_A(x) \leq m_B(x)$ for $x \in U$.

Note that the equality and subset relations in ordinary sets are included in the above definitions for fuzzy sets. In ordinary sets, if an element of set A exists then $m_A(x) = 1$, otherwise $m_A(x) = 0$, in terms of fuzzy sets. If two ordinary sets are equal, and they are interpreted as special cases of fuzzy sets, either $m_A(x) = m_B(x) = 1$ or $m_A(x) = m_B(x) = 0$, depending on whether the element x is in the sets. Similarly, if for ordinary sets $A \subseteq B$, then every element in A is found in B, in which case $m_A(x) = m_B(x) = 1$. For those elements in B but not in A, we have $m_A(x) = 0 < m_B(x) = 1$. We also note that the equality can be expressed in terms of subset relations as in ordinary sets, as: $A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.

"$\in$"**: the element of**: This symbol is not normally used for fuzzy sets. This is because in a fuzzy set, each element is not described as either element ($\in$) or non-element ($\notin$), but by its membership degree. However, "$\in$" can be used for the universe U since the degree of every element in U is 1 as in an ordinary set.

Other relations: The **support** of fuzzy set A is the crisp set of every element in U for which $m_A(x)$ in $A > 0$, i.e., support of $A = \{x \mid m_A(x) > 0, x \in U\}$. For example, if fuzzy set $A = \{x_1/0.1, x_2/0, x_3/0.3, x_4/0, x_5/0.5\}$, then support of $A = \{x_1, x_3, x_5\}$. A **fuzzy singleton** is a fuzzy set whose support is a set of a single element of U. For example, if $A = \{x_1/0, x_2/0.2, x_3/0\}$, then A is a fuzzy singleton since its support is $\{x_2\}$, a set of a single element x_2. If A is a fuzzy singleton whose support is $\{x_0\}$, then $A = \{x_0/m_A(x_0)\}$.

5.2.3 Basic Fuzzy Set Operations and Their Properties

In this subsection, we will define basic fuzzy operations, such as union, which have counterparts in ordinary sets. Fuzzy operations with ordinary counterparts include union, intersection, complement, binary relations and composition of relations. Operations unique to fuzzy sets (i.e., no counterparts in ordinary sets) include fuzzification, and they will be discussed separately in the next section.

There are several different ways of defining these operations. These different ways of definition lead to different properties. For certain types of applications, one definition is convenient, while for some other types of applications, other ways of definitions may be easier for computations that follow. This is why there are different ways of defining these basic operations. In the following, we will discuss the most common definitions. Based on these definitions, we can derive the fuzzy versions of familiar properties in ordinary sets, such as commutative laws, DeMorgan's laws, etc.

Union, intersection, and complement

Let fuzzy set A be represented as $A = \{u/m_A(u) \mid u \in U\}$, where u is an element, $m_A(u)$ the membership function to represent the degree, and U the universe. Here "$u \in U$" means "every u in U."

Union: $A \cup B = \{x/\max(m_A(x), m_B(x)) \mid x \in U\}$
Intersection: $A \cap B = \{x/\min(m_A(x), m_B(x)) \mid x \in U\}$
Complement: $A' = \{x/(1 - m_A(x)) \mid x \in U\}$

Fig. 5.2 shows an example of graphical interpretations of these operations.

Example. Membership functions to be young and old and their union and intersection.

Age: x	Membership function to be young	Membership function to be old	Union	Intersection
≤ 25	1.0	0.0	1.0	0.0
30	0.5	0.0	0.5	0.0
40	0.1	0.2	0.2	0.1
50	0.0	0.6	0.6	0.0
60	0.0	0.8	0.8	0.0
≥ 65	0.0	1.0	1.0	0.0

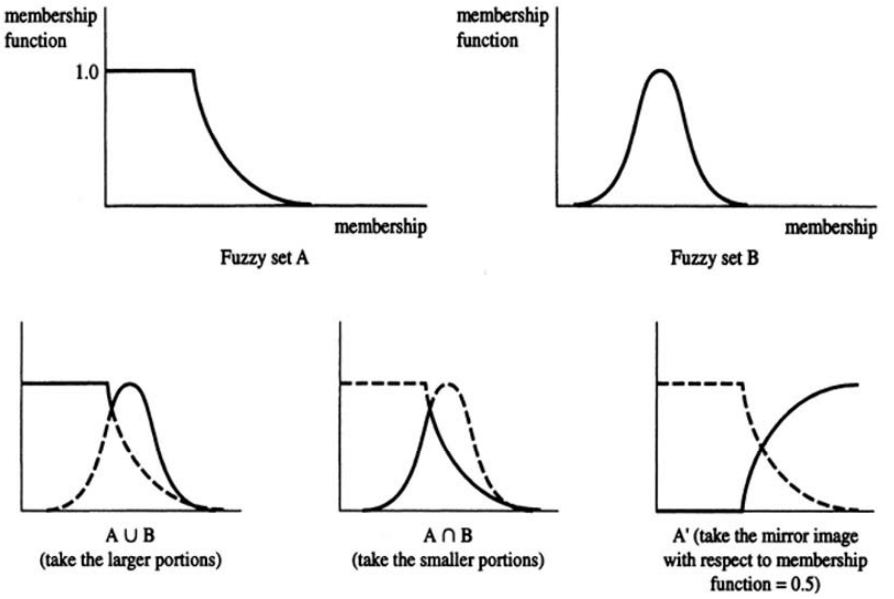


Fig. 5.2. A graphical interpretation of two fuzzy sets and their union, intersection, and complement.

Properties of union, intersection, and complement

Familiar properties for ordinary sets hold as follows.

Commutativity	$A \cup B = B \cup A$ $A \cap B = B \cap A$
Associativity	$(A \cup B) \cup C = A \cup (B \cup C)$ $(A \cap B) \cap C = A \cap (B \cap C)$
Distributive laws	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ $A \cap (\mathrm{B} \cup \mathrm{C}) = (A \cap B) \cup (A \cap C)$
DeMorgan's laws	$(A \cup B)' = A' \cap B'$ $(A \cap B)' = A' \cup B'$
Involution (Double complement)	$(A')' = A$
Idempotent	$A \cup A = A$ $A \cap A = A$

Identity

$A \cup \varnothing = A \qquad A \cap \varnothing = \varnothing$

$A \cup U = U \qquad A \cap U = A$

where $\varnothing$: empty fuzzy set = $\{x/0.0 \mid x \in U\}$ and U: universe = $\{x/1.0 \mid x \in U\}$

Proofs of these properties are straightforward from the definitions of these operations. For example, the commutativity property holds for union, since $A \cup B = \{x/\max(m_A(x), m_B(x)) \mid x \in U\} = \{x/\max(m_B(x), m_A(x)) \mid x \in U\} = B \cup A$.

A fuzzy set as the union of singletons and additional notation for unions, $\cup_U$, Σ, and $\int$

A fuzzy set A can be viewed as the union of its constituent singletons. i.e.,

$$A = \cup_U \{x/m_A(x)\}$$

where $\cup_U$ represents taking the unions over all the elements in U. In particular, when U is finite, we may write

$$A = \Sigma_{i=1,n} \{x_i/m_A(x_i)\}$$

where $n = |U|$ and Σ stands for taking the *unions* (rather than *summation of numbers*) of the argument $\{x_i/m_A(x_i)\}$ over the domain of $i = 1$ to n (i.e., replace "+" in ordinary Σ summation with "$\cup$"). In another particular case, when the elements in U take continuous values, we may replace Σ with $\int$ as in ordinary calculus and write

$$A = \int_U \{x/m_A(x)\}$$

where the integral sign represents taking the unions of the arguments over the domain of U. Note that there is no "dx" at the end of the expression as for integral in ordinary calculus since we are not adding up strips of areas whose width is dx. Remember $\int dx$ for integration, and $\int$ without dx for union in this chapter.

5.2.4 Operations Unique to Fuzzy Sets

In this subsection we define some well known unique operations to fuzzy sets, i.e., operations which have no counterparts in ordinary sets. These operations are discussed here because they often appear in the literature.

Concentration

CON(A) = $\{x/m_A^2(x) \mid x \in U\}$, where $m_A^2(x) = m_A(x) \times m_A(x)$. The following is a graphical interpretation example.

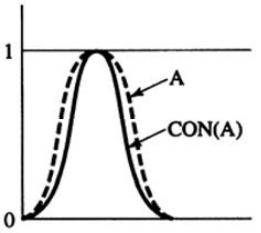


Fig. 5.3. An example of CON(*A*), concentration of set *A*.

Since $0 \le m_A(x) \le 1$, we have $m_A^2(x) < m_A(x)$ except $m_A(x) = 0$ or 1. Hence, CON makes graph narrower and steeper. Furthermore, the "degree of membership reduction" occurs by this operation as:

$$\frac{\left\{m_A(x) - m^2{}_A(x)\right\}}{m_A(x)} = 1 - m_A(x).$$

That is, the elements with low degrees of membership are reduced more than the elements with high degrees. For example, an element with $m_A(x) = 0.1$ is reduced to $m_A{}^2(x) = 0.01$, i.e., 90% or 0.9 decrease, whereas an element with $m_A(x) = 0.9$ is reduced to $m_A^2(x) = 0.81$, i.e., 10% or 0.1 decrease.

Application of concentration. This is a common way of representing "*very* something" as CON("something"), e.g., "very young" = CON("young").

Dilation

$$\mathrm{DIL}(A) = \{x/\sqrt{m_A(x)} \mid x \in U\}.$$

This is the opposite of concentration operation. Elements that are only just barely in the set (e.g., $m_A(x) = 0.01$) increase their degree of membership tremendously (e.g. 10 times to $\sqrt{m_A(x)} = 0.1$). Fig. 5.4 is a graphical illustration example.

Normalization

Normalization of fuzzy set *A*, denoted NORM(*A*), normalizes the membership function in terms of the maximum membership function value. That is, the membership function of *A* is divided by the maximum membership function value to give the membership function of NORM(*A*). The resulting fuzzy set, called the **normal** (or **normalized**) **fuzzy set**, has the maximum membership function value of 1.

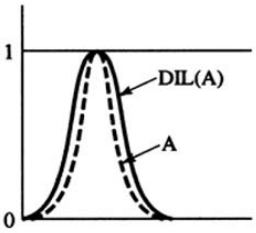


Fig. 5.4 An example of DIL(*A*), dilation of set *A*.

$$\text{NORM}(A) = \{x/\left(\frac{m_A(x)}{\text{Max}}\right) \mid x \in U\},$$

where $\text{Max} = \max_{x \in U} \{m_A(x)\}$. The following is a graphical illustration example.

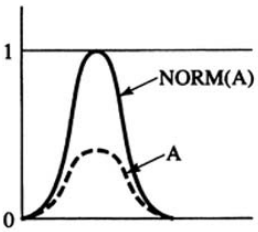


Fig. 5.5 An example of NORM(*A*), normalization of set *A*.

This operation is somewhat analogous to normalization of vectors. For example, normalization of vectors (4, 3) and (8, 6) reduces to the same vector (4/5, 3/5) of length 1. Normalization, in some sense, reduces all fuzzy sets to the same base.

5.3 Fuzzy Relations

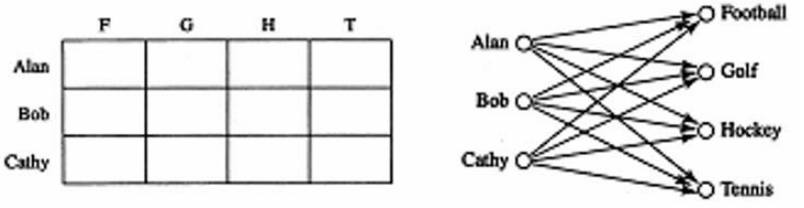
As ordinary sets are extended to fuzzy sets by introducing degrees of elements, ordinary relations can be extended to their fuzzy counterparts. We start with a brief review of ordinary relations before extending the theory to fuzzy relations.

5.3.1 Ordinary (Nonfuzzy) Relations

Cartesian products and relations

Let *A* and *B* be ordinary sets. The **cartesian product** of *A* and *B* is denoted $A \times B$ and

defined as $A \times B = \{(a, b) \mid a \in A, b \in B\}$, where (a, b) is an ordered pair and "$a \in A$, $b \in B$" means that every element of A and B are picked up. Hence, if A has m elements and B has n elements, there will be mn elements in $A \times B$. A **binary relation** (or simply **relation**) R from A to B is any subset of $A \times B$. (For more, see Section 6.2.)



(a) (b)

Fig. 5.6.The cartesian product of the example. (a) Tabular form (F: Football, G: Golf, H: Hockey, T: Tennis) In tabular form, we normally drop "√" from all the entries since they are understood. (b) Diagram form.

Example. A (Person, Sport) relation.

A = {Alan, Bob, Cathy}, B = {Football, Golf, Hockey, Tennis}.

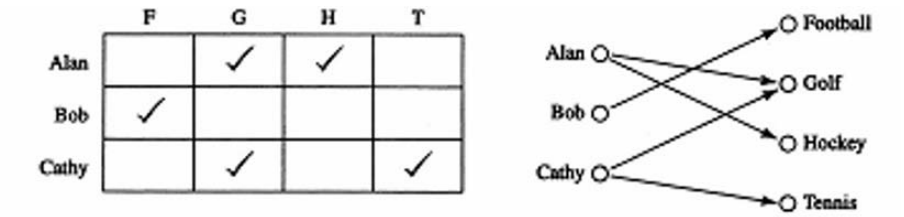
The cartesian product of sets A and B is the set containing $3 \times 4 = 12$ elements as: {(Alan, Football), (Alan, Golf), (Alan, Hockey), (Alan, Tennis), (Bob, Football), ..., (Cathy, Tennis)}. The cartesian product can also be represented by using tabular and diagram forms as in Fig. 5.6. These forms are easier to write and understand than the set form.

Suppose that a relation, R, showing who plays which sports is given as follows:

Alan	Golf, Hockey
Bob	Football
Cathy	Golf, Tennis

Then R is a subset of the cartesian product where: R = {(Alan, Golf), (Alan, Hockey), (Bob, Football), (Cathy, Golf), (Cathy, Tennis)}. The relation can also be conveniently represented by using tabular and diagram forms as in Fig. 5.7.

In general, the cartesian product $A \times B$ includes all possible relations from A to B. In the above example, this means that we consider the set of all possible ordered pairs in form of (Person, Sport). Any relation then is represented by a subset of the cartesian product.



(a) (b)

Fig. 5.7. The relation of the (Person, Sport) relation example. (a) Tabular form. (b) Diagram form.

Compositions of relations

Let A, B, and C be ordinary sets. Let R be a relation from A to B, and let S be a relation from B to C. The **composition** of R and S is a new relation (called the **composite relation**) denoted by R o S and defined by:

$$R \text{ o } S = \{(a, c) \mid (a, b) \in R, (b, c) \in S, a \in A, b \in B, c \in C\}$$

The composition of two relations can be extended to the compositions of three, four, ..., relations as, R_1 o R_2 o R_3, and so on. Since composition operation is associative, that is, $(R_1 \text{ o } R_2) \text{ o } R_3 = R_1 \text{ o } (R_2 \text{ o } R_3)$, we can drop the parentheses and simply write R_1 o R_2 o R_3.

Example. Intelligent computing committee.

The National Research Council has been asked to recommend members of a Presidential Advisory Committee for the next generation of intelligent computing. For simplicity, we consider a much more simplified version of this problem. "Intelligent Information Super-highway (IIS)" and "Intelligent Multimedia (IM)" are selected as key technological areas of intelligent computing. Three prominent scholars, Drs. Adams, Brown and Carter, are among possible candidates for committee members. They are not necessarily experts of the two new technological areas, but very active in more established fields of computer science, such as AI, databases, and networking. Ideal candidates for the committee members do not have to be experts of the technological areas, but must be active in one or more fields in computer science to provide a fair assessment of the new technologies.

In the following, R represents a relation between the scholars and their active computer science fields, and S a relation between the fields and the technological areas. The composite relation R o S then represents how the scholars are indirectly related to the new technological areas through their computer science fields. Figures 5.8 and 5.9 are tabular and diagram forms of these relations, respectively.

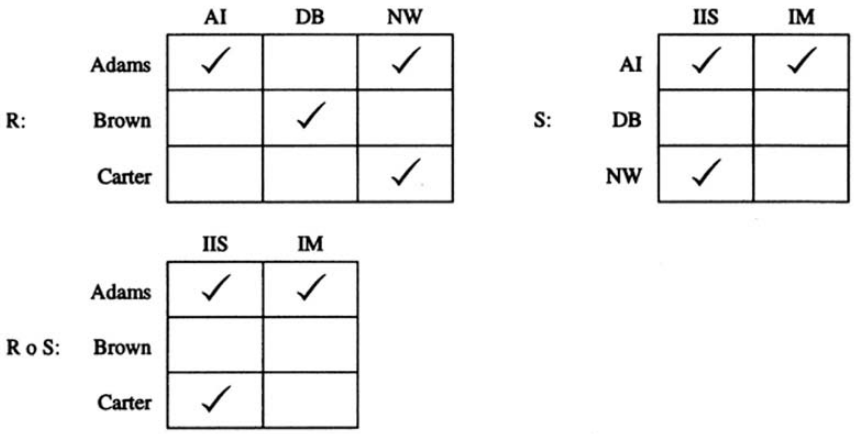


Fig. 5.8.Tabular form of the relations of *R*: (Scholars, Fields), *S*: (Fields, Areas), and the composite of *R* and *S*, *R* o *S*: (Scholars, Areas). DB: Databases, NW: Networking, IIS: Intelligent Information Super-highway, and IM: Intelligent Multimedia.

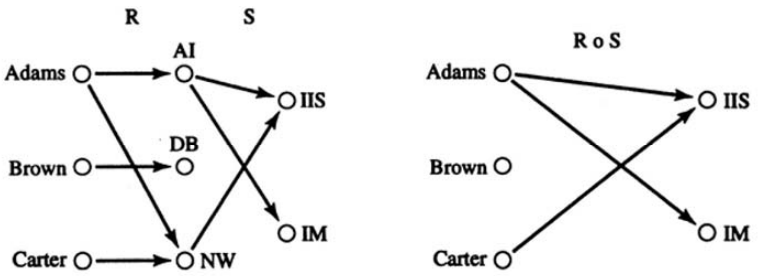


Fig. 5.9.Diagram form of the relations of *R*: (Scholars, Fields), *S*: (Fields, Areas), and the composite of *R* and *S*, *R* o *S*: (Scholars, Areas).

5.3.2 Fuzzy Relations Defined on Ordinary Sets Fuzzy relations

Let A and B be ordinary sets. The cartesian product of $A \times B$ is the one defined previously for ordinary sets as: $A \times B = \{(a, b) \mid a \in A, b \in B\}$. A **fuzzy binary relation** (or simply **fuzzy relation** or **relation**) R from A to B is a fuzzy subset of $A \times B$.

$$R = \{(a, b)/m_R(a, b) \mid a \in A, b \in B\}$$
$$= \cup_{A \times B} \{(a, b)/m_R(a, b)\},$$

where $m_R(a, b)$ is the membership function, and $\cup_{A \times B}$ represents taking the unions of singleton $\{(a, b)/m_R(a, b)\}$ over $A \times B$. In particular, when A and B are finite, and $|A| = m$ and $|B| = n$, we may write,

$$R = \sum_{i=1}^{m}\sum_{j=1}^{n}\left\{\left(a_i, b_j\right) / m_R\left(a_i, b_j\right)\right\},$$

where Σ again stands for taking unions rather than adding numbers. In this case, R can also be represented as a rectangular table called a **relation matrix** by placing $m_R(a_i, b_j)$ as the matrix elements:

$$\begin{bmatrix} m_R(a_1, b_1) & m_R(a_1, b_2) & \ldots & m_R(a_1, b_n) \\ \ldots & & & \\ m_R(a_m, b_1) & m_R(a_m, b_2) & \ldots & m_R(a_m, b_n) \end{bmatrix}$$

When the elements of A and B are continuous, we may write

$$R = \int_A \int_B \{(a, b)/m_R(a, b)\}.$$

In particular, when $A = B$, the cartesian product on A is defined as $A \times A = \{(a, b) \mid a, b \in A\}$, and a fuzzy relation on A is a fuzzy subset of $A \times A$.

Example. A fuzzy (Person, Sport) relation.

In the (Person, Sport) example in the previous subsection, who plays which sports may not be simply either "yes" (1) or "no" (0), but it may be more natural to associate a certain degree to each pair. For example,

	F	*G*	*H*	*T*
Alan	0	0.8	0.7	0.1
Bob	1.0	0	0	0.2
Cathy	0	0.9	0	1.0

This is a relation matrix.

A **fuzzy graph** is a directed graph whose vertices are the elements of the sets under consideration, and whose edges correspond to the elements of the relation, i.e., the ordered pairs. We associate the degree of each ordered pair to each edge as its weight. A fuzzy graph of this example is shown in Fig. 5.10.

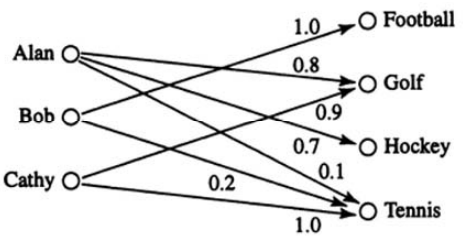


Fig. 5.10. A fuzzy graph for the (Person, Sport) relation.

Example.

Let $X = \{x_1, x_2, x_3\}$, and a fuzzy relation R on X be:

$$R = \{(x_1, x_1)/0.1, (x_1, x_2)/0.2, (x_1, x_3)/0.3, (x_2, x_1)/0.4, (x_3, x_2)/0.5\}.$$

Then the relation matrix is given as follows and a fuzzy graph as in Fig. 5.11.

$$\text{relation matrix} = \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0 & 0 \\ 0 & 0.5 & 0 \end{bmatrix}$$

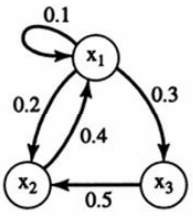


Fig. 5.11. A fuzzy graph for the above example.

k-ary fuzzy relations

As in ordinary relations, binary relations can be extended to ternary, quadruple, and so forth, to *k*-ary relations in general. The cartesian product of *k* crisp sets, $A_1, A_2, ..., A_k$, is represented as $A_1 \times A_2 \times ... \times A_k$, and it is defined as the set of all the *k*-tuples over the domain:

$$A_1 \times ... \times A_k = \{(a_1, ..., a_k) \mid a_1 \in A_1, ..., a_k \in A_k\}.$$

A *k*-ary fuzzy relation is a fuzzy subset of this cartesian product:

$$R = \cup_{A1 \times ... \times Ak} \{(a_1, ..., a_k)/m_R(a_1, ..., a_k)\}.$$

Composition of fuzzy relations

Let *A*, *B*, and *C* be ordinary sets. Let *R* be a fuzzy relation from *A* to *B*, and let *S* be a fuzzy relation from *B* to *C*. The **composition** of *R* and *S* is a fuzzy relation (called the **fuzzy composite relation**) denoted by R o S and defined by:

$$R \text{ o } S = \cup_{A \times C} \{(a, c)/\max_B [\min (m_R(a, b), m_S(b, c))]\},$$

where $\mathbf{max}_B [x]$ means taking the maximum of the x values over all the elements of *B*. If sets *A*, *B*, and *C* are finite, then the relation matrix for R o S is the **max-min**

product of the relation matrices for R and S. The max-min product of two matrices can be determined by replacing *addition by max* and *multiplication by min* in the ordinary matrix multiplication.

Example.

$$\underset{R}{\begin{bmatrix} 0.3 & 0.8 \\ 0.6 & 0.9 \end{bmatrix}} \text{ o } \underset{S}{\begin{bmatrix} 0.5 & 0.9 \\ 0.4 & 1 \end{bmatrix}} = \underset{R \text{ o } S}{\begin{bmatrix} 0.4 & 0.8 \\ 0.5 & 0.9 \end{bmatrix}}$$

For example, 0.4 in R o S is the degree for (a_1, c_1), and it is determined by checking the first row elements of R and the first column elements of S, as is in ordinary matrix multiplication. We compare 0.3 and 0.5 and record 0.3 (the min of these two numbers); we compare 0.8 and 0.4 and record 0.4 (the min of these two numbers). Then we take max of these recorded numbers, 0.3 and 0.4, obtaining the answer 0.4. This process is a special case of the above general formula for R o S as: $\max_B$ [min $(m_R(a_1, b), m_S(b, c_1))$] = max {[min $(m_R(a_1, b_1), m_S(b_1, c_1))$], [min $(m_R(a_1, b_2), m_S(b_2, c_1))$]} = max {[min (0.3, 0.5)], [min (0.8, 0.4)]} = max {0.3, 0.4} = 0.4.

As a special case, the matrix R may have only one row, in which case R is called a vector. When R is a vector, R o S is also a vector.

Example.

$$\underset{R}{\begin{bmatrix} 0.2 & 1 \end{bmatrix}} \text{ o } \underset{S}{\begin{bmatrix} 0.8 & 0.9 \\ 0.6 & 0.7 \end{bmatrix}} = \underset{R \text{ o } S}{\begin{bmatrix} 0.6 & 0.7 \end{bmatrix}}$$

Example. A fuzzy version of the intelligent computing committee.

In the previous Presidential Advisory Committee example, it is natural to extend R and S to be fuzzy relations, representing the degrees of their relations as follows. The fuzzy composite relation, R o S, then can be determined as follows. Figures 5.12 and 5.13 are matrix and diagram forms of these relations, respectively.

We note that the max-min operation for the fuzzy composition operation in the above example may be a simple and reasonable approach. For example, Adams is related to Intelligent Information Super-highway through two routes: Adams (0.7) AI (0.9) IIS and Adams (0.8) Networking (0.5) IIS. We first take the minimum in each route, since the strength of the route is limited to by the minimum. We then take the maximum among the strengths of the alternative routes, since all routes are possible.

$$R: \begin{array}{c} \\ \text{Adams} \\ \text{Brown} \\ \text{Carter} \end{array} \begin{array}{c} \begin{array}{ccc} \text{AI} & \text{DB} & \text{NW} \end{array} \\ \begin{bmatrix} 0.7 & 0 & 0.8 \\ 0 & 1.0 & 0 \\ 0 & 0 & 0.9 \end{bmatrix} \end{array}$$

$$S: \begin{array}{c} \\ \text{AI} \\ \text{DB} \\ \text{NW} \end{array} \begin{array}{c} \begin{array}{cc} \text{IIS} & \text{IM} \end{array} \\ \begin{bmatrix} 0.9 & 0.6 \\ 0 & 0.1 \\ 0.5 & 0 \end{bmatrix} \end{array}$$

$$R \circ S: \begin{array}{c} \\ \text{Adams} \\ \text{Brown} \\ \text{Carter} \end{array} \begin{array}{c} \begin{array}{cc} \text{IIS} & \text{IM} \end{array} \\ \begin{bmatrix} 0.7 & 0.6 \\ 0 & 0.1 \\ 0.5 & 0 \end{bmatrix} \end{array}$$

Fig. 5.12.Matrix form of the relations of *R*: (Scholars, Fields), *S*: (Fields, Areas), and the composition of *R* and *S*, *R* o *S*: (Scholars, Areas).

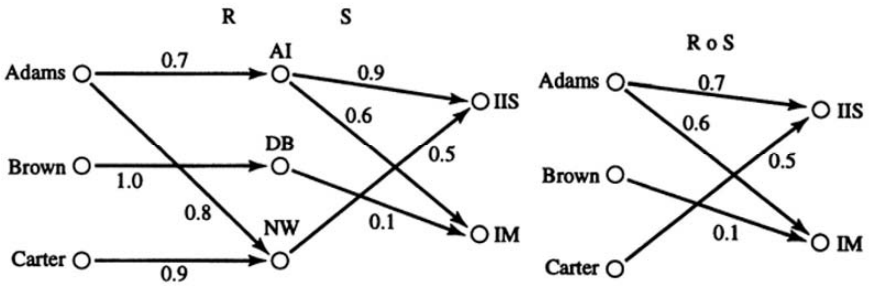


Fig. 5.13.Diagram form of the relations of *R*: (Scholars, Fields), *S*: (Fields, Areas), and the composition of *R* and *S*, *R* o *S*: (Scholars, Areas).

There are, however, no universally accepted operations for fuzzy composition operation. If, for example, the max-min operation is not appropriate for certain applications, we can employ some other form of operation. In fact, other forms of fuzzy composition operations have been proposed. For example, min(a, b) may be replaced by ($a \times b$). This is similar to computing the probability that two mutually independent events occur. (For example, the probability for the first coin tossing is Head and the second tossing is also Head is $0.5 \times 0.5 = 0.25$.) The max(a, b)

operation may be replaced by $\min((a + b), 1)$. Other forms of operations are also possible. The selection of operations is determined by the appropriateness for a specific application and the simplicity.

5.3.3 Fuzzy Relations Derived from Fuzzy Sets

Let A be a fuzzy set of a universe U, and B be a fuzzy set of a universe V. The cartesian product $A \times B$ can be defined by:

$$A \times B = \{(u, v)/\min(m_A(u), m_B(v)) \mid u \in U, v \in V\}$$
$$= \cup_{U \times V} \{(u, v)/\min(m_A(u), m_B(v))\}.$$

Note that when A and B are ordinary sets where $m_A(u)$ and $m_B(v)$ are either 0 or 1, the above $A \times B$ reduces to the ordinary cartesian product. $A \times B$ is a fuzzy relation from U to V. A fuzzy relation R from A to B can be defined as $R = \{(u, v)/m(u, v) \mid m(u, v) \leq m_A(u), m(u, v) \leq m_B(v), \mid u \in U, v \in V\}$.

The above definitions can be extended to k-ary relations. Let A_1 be a fuzzy set of a universe U_1, A_2 be a fuzzy set of a universe U_2, ..., and let A_k be a fuzzy set of a universe U_k. The cartesian product $A_1 \times A_2 \times ... \times A_k$ can be defined by:

$$A_1 \times A_2 \times ... \times A_k$$
$$= \{(u_1, u_2, ..., u_k)/\min(m_{A1}(u_1), m_{A2}(u_2), ..., m_{Ak}(u_k)) \mid u_1 \in U_1, u_2 \in U_2, ..., u_k \in U_k\}$$
$$= \bigcup_{U1 \times U2 \times ... \times Uk} \{(u_1, u_2, ..., u_k)/\min(m_{A1}(u_1), m_{A2}(u_2), ..., m_{Ak}(u_k))\}.$$

A fuzzy k-ary relation can be extended similarly.

5.4 Fuzzy Logic

5.4.1 Ordinary Set Theory and Ordinary Logic

There are similarities between ordinary (nonfuzzy) set theory and ordinary (nonfuzzy) logic. The following table shows correspondences between these two fields.

Ordinary Set	Ordinary Logic
existence of an element	true
non-existence of an element	false
$\cap$	AND
$\cup$	OR
complement	NOT

In ordinary propositional calculus, let A and B be propositional variables. **Implication** is denoted as $A \Rightarrow B$, and read "A implies B" or "if A then B".

5.4.2 Fuzzy Logic Fundamentals

As fuzzy sets are extensions of ordinary sets, *fuzzy logic* is an extension of ordinary logic. As there are similarities between ordinary sets and ordinary logic, so are between fuzzy set theory and fuzzy logic. The following table shows these correspondences.

Fuzzy Set	Fuzzy Logic
degree of membership	truth value of proposition
$\cap$	AND
$\cup$	OR
complement	NOT

Fuzzy implication: "if *A* then *B*" and "if *A* and *B* then *C*"

$A \Rightarrow B$, i.e., "A implies B" or "if A then B" where A and B are fuzzy sets rather than ordinary propositional variables. For example, "if x is young then y is small" or simply "if young then small" is a fuzzy implication. A fuzzy implication is viewed as describing a relation between two fuzzy sets. There are different ways of defining a fuzzy implication as a relation. But there is no standard definition, and the choice depends on the type of application. One common definition, particularly used in fuzzy control, is

$$A \Rightarrow B = A \times B$$

where $A \times B$ is the cartesian product of fuzzy sets A and B. That is, as discussed before, suppose that A is a fuzzy set of a universe U, and B is a fuzzy set of a universe V. Then the cartesian product $A \times B$ is:

$$\begin{aligned} A \times B &= \{(u, v)/\min(m_A(u), m_B(v)) \mid u \in U, v \in V\} \\ &= \bigcup_{U \times V} \{(u, v)/\min(m_A(u), m_B(v))\}. \end{aligned}$$

An extension of the above is (A and B) $\Rightarrow C$, i.e., "if A and B then C." A common definition of this fuzzy implication, particularly used in fuzzy control, is $A \times B \times C$:

$$\begin{aligned} A \times B \times C &= \{(u, v, w)/\min(m_A(u), m_B(v), m_C(w)) \mid u \in U, v \in V, w \in W\} \\ &= \bigcup_{U \times V \times W} \{(u, v, w)/\min(m_A(u), m_B(v), m_C(w))\}. \end{aligned}$$

Compositional rule of inference

Let R be a fuzzy relation from U to V, X be a fuzzy subset of U, Y be a fuzzy subset

of V, and $Y = X$ o R. Y is said to be **induced** by X and R. In the above, "o" represents the composition of X and R. Here X and Y can be viewed as row vectors whose components are the values of the membership functions as:

$$X = \{u/m_X(u) \mid u \in U\}$$
$$Y = \{v/m_Y(v) \mid v \in V\}$$

As discussed before, X o R is the max-min product of the vector X and the relation matrix R:

$$X \text{ o } R = \{v / [\max (\min (m_X (u), m_R (u, v))) \mid u \in U] \mid v \in V\}$$
$$= \bigcup_V \{v / \max_U [\min (m_X(u), m_R(u, v))]\}$$

Example.

$$\underset{X}{[0.2\ 1\ 0.3]} \text{ o } \underset{R}{\begin{bmatrix} 0.8 & 0.9 & 0.2 \\ 0.6 & 1 & 0.4 \\ 0.5 & 0.8 & 1 \end{bmatrix}} = \underset{Y}{[0.6\ 1\ 0.4\ ]}$$

Fuzzy Inference

1. "if A then B" and $X \rightarrow Y$

Fuzzy inference is based on fuzzy implication and the compositional rule of inference discussed above. The basic steps are as follows. Let A and X be fuzzy sets of a universe of U, B and Y be fuzzy sets of a universe V. Suppose that we are given:

a) Implication: "if A then B"
b) Premise: X is true

and we want to determine:

c)Conclusion: Y.

To achieve the above, perform the following two steps:

Step 1. Compute the fuzzy implication, "if A then B," as a fuzzy relation $R = A \times B$.
Step 2. Induce Y by $Y = X$ o R.

Using the membership functions for A, B, and X, we can compute the membership function for Y as follows:

$$Y = X \text{ o } R$$

$$= X \text{ o } (A \times B)$$
$$= \{v / [\max (\min (m_X(u), m_A(u), m_B(v))) \mid u \in U] \mid v \in V\}$$
$$= \bigcup_V \{v / \max_U [\min (m_X(u), m_A(u), m_B(v))]\}.$$

2. "if A and B then C" and X and $Y \rightarrow Z$

An extension of the above is (A and B) => C, i.e., "if A and B then C," X and Y are true, then derive conclusion Z. Let A and X be fuzzy sets of a universe of U, B and Y be fuzzy sets of a universe V, and C and Z be fuzzy sets of a universe of W. Then Z is computed as follows:

$$Z = (X \times Y) \text{ o } (A \times B \times C)$$
$$= \{w / [\max (\min (m_X(u), m_Y(v), m_A(u), m_B(v), m_C(w))) \mid u \in U, v \in V] \mid w \in W\}$$
$$= \bigcup_W \{w / \max_{U \times V} [\min (m_X(u), m_Y(v), m_A(u), m_B(v), m_C(w))]\}.$$

Example. "if A then B" and $X \rightarrow Y$

Let the universe: $U = \{1, 2, 3, 4, 5\}$. We define two fuzzy sets:

small	=	{1/1, 2/0.8, 3/0.6, 4/0.4, 5/0.2}
large	=	{1/0.2, 2/0.4, 3/0.6, 4/0.8, 5/1}

From these two fuzzy sets, we can derive other fuzzy sets as, for example:

not large	=	{1/0.8, 2/0.6, 3/0.4, 4/0.2}
very small	=	{1/1, 2/0.64, 3/0.36, 4/0.16, 5/0.04}
not very small	=	{2/0.36, 3/0.64, 4/0.84, 5/0.96}

Here the membership function of "very" X is computed from the membership function of X, $m_X(u)$, by $\{u/m_X^2(u) \mid u \in U\}$. Now let our problem be described as follows:

Our implication:	if A then B, where A = small and B = large.
Our premise:	X = not large.

What can we conclude for Y?

In this example, we first compute R by $A \times B$, then determine the answer Y by $Y = X$ o R.

Step 1. Derive R from "if A then B" by $A \times B$, where A = small and B = large:

$$R = \begin{bmatrix} 0.2 & 0.4 & 0.6 & 0.8 & 1 \\ 0.2 & 0.4 & 0.6 & 0.8 & 0.8 \\ 0.2 & 0.4 & 0.6 & 0.6 & 0.6 \\ 0.2 & 0.4 & 0.4 & 0.4 & 0.4 \\ 0.2 & 0.2 & 0.2 & 0.2 & 0.2 \end{bmatrix}$$

Step 2. Induce Y by $Y = X$ o R, where X = not large:

$$Y = \underset{X}{[0.8\ 0.6\ 0.4\ 0.2\ 0]} \text{ o } \underset{R}{\begin{bmatrix} 0.2 & 0.4 & 0.6 & 0.8 & 1 \\ 0.2 & 0.4 & 0.6 & 0.8 & 0.8 \\ 0.2 & 0.4 & 0.6 & 0.6 & 0.6 \\ 0.2 & 0.2 & 0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 & 0.2 & 0.2 \end{bmatrix}}$$

$$= [0.2\ \ 0.4\ \ 0.6\ \ 0.8\ \ 0.8]$$

In the above inference, we followed the basic steps of fuzzy inference discussed earlier to compute $R = A \times B$, then to determine Y by X o R. We can also obtain the same result for Y by using the following formula, which discussed after the basic steps.

$$\begin{aligned} Y &= X \text{ o } R \\ &= X \text{ o } (A \times B) \\ &= \bigcup_V \{v / \max_U [\min (m_X(u), m_A(u), m_B(v))]\}. \end{aligned}$$

Note that as a special case of the fuzzy inference discussed here, when $X = A$ and A is a *normal fuzzy set* (i.e., the maximum degree of A is 1), we can show that $Y = X$ o $R = X$ o $(A \times B) = A$ o $(A \times B)$ is equal to B as follows:

$$\begin{aligned} Y &= A \text{ o } (A \times B) \\ &= \{v / [\max (\min (m_A(u), m_A(u), m_B(v))) \mid u \in U] \mid v \in V\} \\ &= \{v / [\max (\min (m_A(u), m_B(v))) \mid u \in U] \mid v \in V\} \\ &= \bigcup_V \{v / \max_U [\min (m_A(u), m_B(v))]\} \\ &= \bigcup_V \{v / [\min (\max_U (m_A(u)), m_B(v))]\} \\ &= \bigcup_V \{v / [\min (1, m_B(v))]\} \qquad \text{(since } A \text{ is normal)} \end{aligned}$$

$$= \bigcup_{V} \{v / m_B(v)\}$$
$$= B.$$

5.5 Fuzzy Control

Control refers to the control of various physical, chemical, or other numeric characteristics, such as temperature, electric current, flow of liquid or gas, motion of machines, various business and financial quantities (e.g., flow of cash, inventory control), and so forth. A control system can be abstracted as a box for which inputs are flowing into it, and outputs are emerging from it. Parameters can be included as parts of inputs or within the box, i.e., the control system.

For example, consider a system that controls some kind of temperature distribution by heat and possibly cooling sources. The inputs may be the current temperature distribution and its time derivatives, and a parameter may be target temperature distribution. The outputs can be the amounts of the heat and cooling sources to be applied. The control problem in general is to develop a formula or algorithm for mapping from the inputs and parameters to the outputs.

Fuzzy control is a control technique based on fuzzy logic. Given input, typically system measurements such as temperature, we are to determine output such as an amount of the heat source to control the system. In fuzzy control, the rules, input, and/or output may involve fuzziness, leading to the use of fuzzy logic.

The basic idea of fuzzy control is to apply fuzzy inference to control problems. In fuzzy control, the control box includes fuzzification, fuzzy inference using fuzzy if-then rules, and defuzzification procedures. Fuzzy rules can include human descriptive judgements, such as "if the temperature is moderately high and the pressure is very low, then the output is medium." Although fuzzy control is based on fuzzy inference, simple methods are used in considering computation time.

5.5.1 Fuzzy Control Basics

In a fuzzy control system, we have a set of **fuzzy control rules** in the format of "if x = small, then z = big," or "if x = small and y = medium, then z = big." Here x and y are the **input variables** and z is the **output variable**. Given specific values of x and y, our task is to determine a value of z using applicable control rules and fuzzy inference.

Commonly used fuzzy variables and their membership functions

We define **fuzzy variables** that can represent values of the input and output variables. A commonly used set of seven fuzzy variables follows:

NB = Negative Big
NM = Negative Medium
NS = Negative Small
ZO = Zero
PS = Positive Small
PM = Positive Medium
PB = Positive Big

Or, the two mediums, *NM* and *PM*, may be omitted, resulting in the following set of five fuzzy variables. This smaller set of fuzzy variables is simpler, but it would result less fine or delicate control. For simplicity, we will use this five fuzzy variable version hereafter.

NB = Negative Big
NS = Negative Small
ZO = Zero
PS = Positive Small
PB = Positive Big

The next step is to define membership functions for these fuzzy variables. Defining a membership function is up to us, and the selection of membership functions affects the control performance. What membership function we choose depends on many factors, such as the type of application, how much fine control is required, how fast the control must be performed, and so on. A rule of thumb is that simpler membership function causes lesser computation time but reduces fine control.

There are two categories of membership functions. One is continuous and the other discrete. The following, Fig. 5.14, shows an example of the continuous membership function. In this example, each fuzzy variable's membership function has a triangular shape (plus the zero membership function outside of the triangle, i.e., the bottom line segments). There are other common continuous membership functions such as trapezoids (rather than triangles) and bell shapes.

In Fig. 5.14, the membership function value or degree of variable *PS* is 1 when normalized $x = 0.5$; *PS* is 0.5 when normalized $x = 0.25$ or 0.75; and *PS* is 0 when $x = 1$ or ≤ 0. Mathematically, a triangular membership function $m(x)$ can be represented as:

$$m(x) = \max\left[\frac{\left(a - |x-b|\right)}{a}, 0\right],$$

where $a > 0$ and b are constants. b determines the x value for the apex or the symmetric point, and $2a$ represents the width of the triangle base. For example, in Fig. 5.14, for the membership function for variable *NS*, we choose $a = 2$ and $b = -2$ for not normalized, and $a = 0.5$ and $b = -0.5$ for normalized.

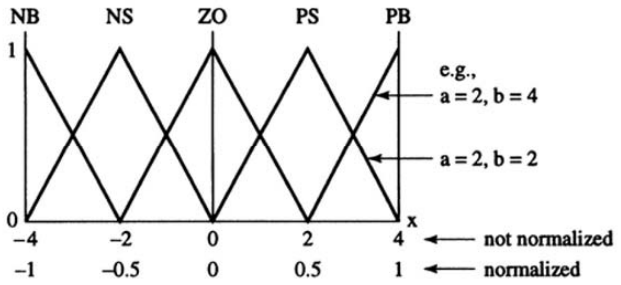


Fig. 5.14. Graphical representations of continuous ***triangular membership functions for five fuzzy variables***.

The following is an example of discrete membership functions.

Table representations of discrete membership functions for five fuzzy variables

x	-4	-3	-2	-1	0	1	2	3	4
$m(x)$:									
NB	1	0.5	0	0	0	0	0	0	0
NS	0	0.5	1	0.5	0	0	0	0	0
ZO	0	0	0	0.5	1	0.5	0	0	0
PS	0	0	0	0	0	0.5	1	0.5	0
PB	0	0	0	0	0	0	0	0.5	1

Example.

Assume that acceleration of a space shuttle is between -4*G* and +4*G* (*G* represents the gravitational acceleration). Using five fuzzy variables, the triangular membership function is represented by Fig. 5.14. A negative acceleration of -1.5 *G*, that is, the normalized value of -0.375, for example, is represented as, *NS* with the degree of 0.75, and *ZO* with the degree of 0.25 (and *NB*, *PS*, and *PB* with the degree of 0).

Typical fuzzy control setup

We will describe a typical fuzzy control setup in the following. At each time interval our fuzzy control system receives specific values for two inputs, E and ΔE, and yields one output, W:

$$E \text{ and } \Delta E \rightarrow \text{Fuzzy control system} \rightarrow W$$

That is, E and ΔE are our input variables, and W is our output variable. For example, at some specific time, E may be 3 and ΔE may be 0, and then W may be determined as -2.2. After a short time interval, the values of E and ΔE change, and a new value of W is computed. This process continues over a certain time period until control has been achieved.

Suppose that T represents the value to be controlled by the system. If we are to

control temperature, T will be the temperature; to control speed, T will be the speed, etc. Then T_0 is the target T, and E is the difference between T and T_0.

$E = T - T_0$: the general expression for temperature difference

Since T and E change step by step over time, we can define T and E at each time period n as T_n and E_n, respectively.

$E_n = T_n - T_0$: the temperature difference at time period n

Then ΔE is defined as follows (ΔE represents the time derivative of E at time period n).

$\Delta E = E_n - E_{n-1}$: the changing rate of E at time period n

Just as E is the difference between the current and target values, rather than the current control value itself, W is a deviation from the current output value. For example, suppose that temperature is controlled by a heat source, and the amount of heat at time period n is Z_n; then

$$Z_{n+1} = Z_n + W.$$

Note. In some books, E is defined as $E = T_0 - T$, i.e., the sign will be reversed. Using this definition, the sign of ΔE is also reversed: $\Delta E = E_n - E_{n-1} = (T_0 - T_n) - (T_0 - T_{n-1}) = -(T_n - T_{n-1})$. In the fuzzy if-then table discussed below, the terms of "Negative" and "Positive" for E and ΔE will be interchanged (e.g., replace *NB* with *PB*). The table entries for W remain the same for this definition of E.

Fuzzy if-then rules that derive W from E and ΔE

We set up a table for fuzzy if-then rules that derive W from E and ΔE in terms of the fuzzy variables. The following is such a table.

Fuzzy if-then rule table for $(E, \Delta E) \rightarrow W$

W		ΔE *NB*	*NS*	*ZO*	*PS*	*PB*
	NB			*PB* ← (Rule 6)		
	NS			*PS*		
E	*ZO*	*PB*	*PS*	*ZO*	*NS*	*NB*
	PS	↑		*NS* ← (Rule 8)		
	PB	(Rule 1)		*NB* ← (Rule 9)		

This table represents nine rules corresponding to the nine entries in the table. For example, "if $E = ZO$ and $\Delta E = NB$, then $W = PB$" may be called Rule 1. The remaining four entries in the same horizontal line of the table may be called Rules 2, 3, 4 and 5. The remaining four entries in the vertical line may be called Rules 6, 7, 8 and 9. That

is,

Rule 1: if E is ZO and ΔE is NB then W is PB,
or

Rule 2: if E is ZO and ΔE is NS then W is PS,
or
:
:

Rule 8: if E is PS and ΔE is ZO then W is NS,
or

Rule 9: if E is PB and ΔE is ZO then W is NB.

System response phases

System response in fuzzy control, i.e., the behavior of the value to be controlled with respect to time, is shown in Fig. 5.15. The response is an oscillating (irregular) cycles with decaying amplitude, where each cycle consists of four phases, I through IV. The Phase I of Cycle 1, i.e., the beginning of control, is near point a_1 in the figure, where E has the most negative value and ΔE is near zero. Hence, Rule 6 in the if-then rule table applies to the region around point a_1. Around point b_1, E is near zero and ΔE is a large positive number. Hence, Rule 5 in the if-then rule table applies to this point. Similarly, around point c_1, E is large and positive and ΔE is near zero, so Rule 9 applies; around point d_1, E is near zero and ΔE is most negative, and Rule 1 applies to this point. After Phases I, II, III, and IV of Cycle 1, the four phases repeat for Cycle 2, with smaller amplitude. Hence, near point a_2, E is small and negative, ΔE is near zero, and Rule 7 applies.

The following fuzzy if-then rule table shows some points in Fig. 5.15 and their major corresponding rules. As we will see later in a case study, typically more than one rule is applied to a point, which is a major feature of fuzzy systems in general. Points a_3, b_3, c_3, and so on will continue in the same fashion, forming a shrinking spiral in the table. Whether a_3 or any subsequent point converges as a major rule to Rule 3, where $E = ZO$ and $\Delta E = ZO$, depends on how fast the amplitude decreases. Perhaps a_3 stays on Rule 7, the same rule as for a_2, and b_3 converges to Rule 3, $E = ZO$ and $\Delta E = ZO$.

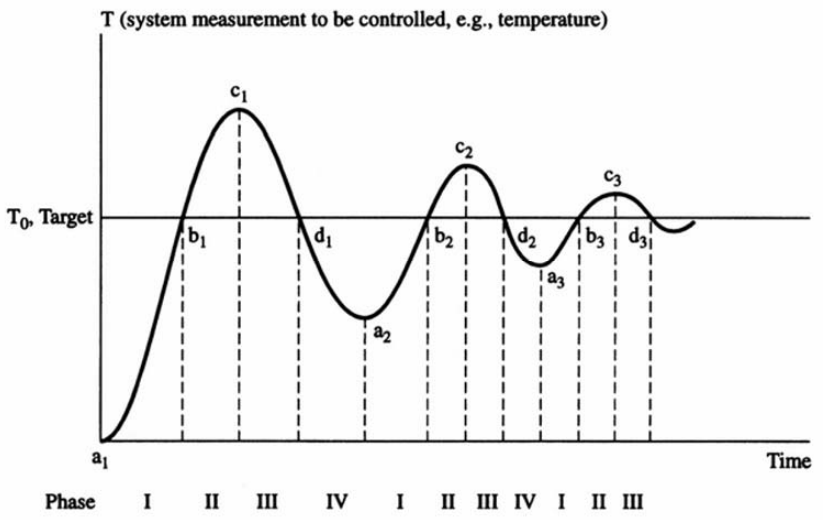


Fig. 5.15 System response in fuzzy control

				ΔE		
W		*NB*	*NS*	*ZO*	*PS*	*PB*
	NB			*PB* a_1		
	NS			*PS* a_2		
E	*ZO*	*PB* d_1	*PS* d_2	*ZO*	*NS* b_2	*NB* b_1
	PS			*NS* c_2		
	PB			*NB* c_1		

In the fuzzy if-then rule table, the nine entries shown, which represent the phases of different cycles, are considered the major entries required to achieve the required control. The empty entries of the table (such as $E = NB$ and $\Delta E = NB$) are considered not important for several reasons. First, for fuzzy control as depicted in Fig. 5.15, cases in which both E and ΔE take extreme values (e.g., $E = NB$ and $\Delta E = NB$; $E = PB$ and $\Delta E = NB$) do not occur. That is, when one of E and ΔE takes an extreme value, the other is near zero. Second, as we will see soon, this if-then rule table is used in conjunction with membership functions like Fig. 5.14. Each value of a fuzzy variable (e.g., $\Delta E = ZO$) does not represent a single point, but instead covers a wide range (e.g., ZO covers x = -2 to 2) with varying degrees. That is, each rule in the table covers wide ranges of E and ΔE. A third reason is, as we will see in the case study in the next subsection, W is typically chosen as a deviation (a small additive term) from the current system output, rather than the system output itself. For such W, the effect of W is not as critical as the system output itself.

These reasons allow us to have fewer rules to perform the required control, and normally, simple assumptions are made for these empty entries. For example, assume $W = 0$ (not $W = ZO$) for all these empty entries. When W represents a deviation from

the current system output, $W = 0$ means to keep the current system output. In certain applications, some of the empty entries are filled in, as for example: $E = PS$ and $\Delta E = NB$ then $W = NS$, $E = PB$ and $\Delta E = NS$ then $W = PS$, $E = PS$ and $\Delta E = NS$ then $W = ZO$, etc.

A cookbook recipe to compute output, *W*, from two inputs, *E* and ΔE

Now with all this predetermined information, we can compute W for the given values of E and ΔE.

1. *Fuzzification.*
   Look at Fig. 5.14 and find which fuzzy variables (*NB*, etc.) apply to the given specific value of E (x in the figure) and to what degree. Repeat the same for ΔE.

2. *Fuzzy inference.*

   a) Look at the fuzzy if-then rule table above and find which rules apply for the fuzzy variable combinations found for E and ΔE in Step 1. Let call these rule numbers i and j (e.g., if Rules 8 and 9 are applicable, then $i = 8$ and $j = 9$).

   b) Compute the **weight** (**firing strength**), α_i, of each rule found in Substep (a) in the form of

   $$\alpha_i = \min(m_{F1}(E), m_{F2}(\Delta E)) = m_{F1}(E) \wedge m_{F2}(\Delta E)$$

   where $F1$ and $F2$ are the fuzzy variables found in Step 1, and $\wedge$ takes the minimum of the operand membership functions.

   c) Find the membership function for W associated with each rule in the form of

   $$m_i(W) = \alpha_i \wedge m_{F3}(W)$$

   where $F3$ is the fuzzy variable found for W in the fuzzy if-then rule table, and $m_{F3}(W)$ is the membership function corresponding to that fuzzy variable in Fig. 5.14 (let x be W).

   Note that in these substeps, we employ the fuzzy implication formula in the form of "if E and ΔE then W" $= E \times \Delta E \times W$, taking the minimum of the membership functions of these variables, E, ΔE, and W.

   d) Compute the membership function for W, $m_T(W)$, in the following form:

   $$m_T(W) = \max(m_i(W), m_j(W)) = m_i(W) \vee m_j(W),$$

where $\vee$ takes the maximum of the operand membership function. Note that since these rules are combined in the form of Rule *i or* Rule *j*, we take the *max* of these membership functions.

3. *Defuzzification*.

The above $m_T(W)$ gives the fuzzy version of the solution, i.e., the answer for output W as a (membership) function of W. (For example, an answer may be to produce output W in the form of -4 to -1 with a degree of 0.5, and so on.) For practical output, however, we need a specific single value, W_0, as a system output to perform the control. For this purpose, we compute the "mean" of W weighted by $m_T(W)$ or the "center of gravity" of $m_T(W)$ as W_0 as follows:

$$W_0 = \frac{\int W \cdot m_T(W)\, dW}{\int m_T(W)\, dW}$$

Here the $\int$ symbol represents ordinary integration, rather than fuzzy union (note "dW" at the end of the expression). This process of evaluating the center of gravity is called a **defuzzification procedure**.

5.5.2 Case Study: Controlling Temperature with a Variable Heat Source

In the following, we will illustrate the basic procedure of fuzzy control discussed above by using a simple example. Our case study problem is described below.

Problem description

We have one type of system measurement, temperature, T. Let T_n be the temperature at time period n, and T_0 be the target temperature. From these values, we compute the following:

$E = T - T_0$:	general expression for temperature difference
$E_n = T_n - T_0$:	temperature difference at time period n
$\Delta E = E_n - E_{n-1}$:	changing rate of E at time period n

We also have one type of system output, the changing rate of heat source, W. W represents a small difference from the current heat source. If the current heat source is Z_n, then $Z_{n+1} = Z_n + W$.

Our case study problem is as follows: given two input values, E = (the difference between the current temperature and the target temperature) and ΔE = (the time derivative of the difference), we are to determine output value, W = (the changing rate of heat source).

Suppose that (not normalized) $E = 3$ and $\Delta E = 0$.

Step 1. *Fuzzification.*

By looking at the not normalized $x = 3$ in Fig. 5.14 (for $E = 3$), we find the membership function of E is *PB* (Positive Big) with the degree = 0.5 and *PS* with degree = 0.5. Similarly, the membership function of ΔE is *ZO* with the degree = 1.0.

Step 2. *Fuzzy inference*

a) Hence, in the fuzzy if-then rule table, two rules are applicable: Rules 8 and 9. (Rule 8: if E is PS and ΔE is ZO, then W is NS; Rule 9: if E is PB and ΔE is ZO, then W is NB.)

b) For each of these two rules, we compute the weight (firing strength):

$$\alpha_8 = m_{PS}(E) \wedge m_{ZO}(\Delta E) = 0.5 \wedge 1.0 = 0.5$$
$$\alpha_9 = m_{PB}(E) \wedge m_{ZO}(\Delta E) = 0.5 \wedge 1.0 = 0.5$$

c) We find the membership function for W associated with each rule as:

$$m_8(W) = \alpha_8 \wedge m_{NS}(W) = 0.5 \wedge m_{NS}(W) \quad \text{(see Fig. 5.16)}$$
$$m_9(W) = \alpha_9 \wedge m_{NB}(W) = 0.5 \wedge m_{NB}(W) \quad \text{(see Fig. 5.17)}$$

d) The membership function for W, $m_T(W)$, is obtained as the max of the above two intermediate membership functions, $m_8(W)$ and $m_9(W)$ (see Fig. 5.18).

$$m_T(W) = m_8(W) \vee m_9(W).$$

Step 3. *Diffuzification*.

We compute the center of gravity of $m_T(W)$ as W_0 (see Fig. 5.19). The numerator of $W_0 = \int W \cdot m_T(W)\, dW = \int_{-4}^{-1} W \cdot (0.5)\, dW + \int_{-1}^{0} W \cdot (-W/2)\, dW = (0.5)W^2/2 \,|_{-4}^{-1} + (-0.5)\, W^3/3 \,|_{-1}^{0} = -47/12$. The denominator of $W_0 = \int m_T(W)\, dW = \int_{-4}^{-1} (0.5)\, dW + \int_{-1}^{0} (-W/2)\, dW = (0.5)W \,|_{-4}^{-1} + (-0.5)\, W^2/2 \,|_{-1}^{0} = 7/4$. Hence, $W_0 = (-47/12)/(7/4) = -2.2381$.

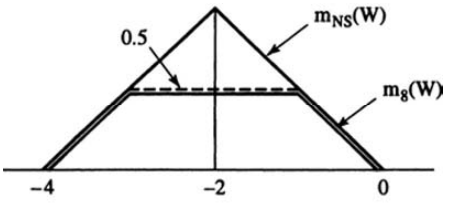


Fig. 5.16 $m_8(W) = 0.5 \wedge m_{NS}(W)$

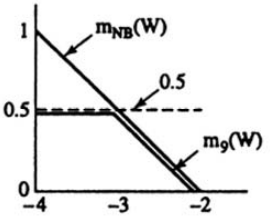


Fig. 5.17 $m_9(W) = 0.5 \wedge m_{NB}(W)$

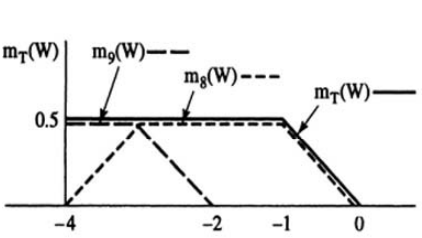


Fig. 5.18. the membership function for W, $m_T(W) = \max(m_8(W), m_9(W))$

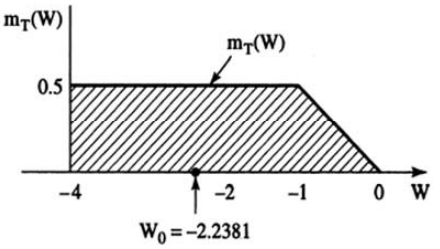


Fig. 5.19. Final output, W_0, as the center of gravity of $m_T(W)$

Possible extensions of this simple case study are multiple system measurements (rather than a single measurement of temperature) and/or outputs (rather than a single output of W).

Programming considerations

Writing a short program for a simple problem such as the case study discussed here is a good way to understand the basics of fuzzy control. Integrations may be carried out by using simple formula. One such simple formula would be to add up narrow rectangular areas, where each rectangle has width dW. Or, when the membership functions for fuzzy variables are triangular as in our discussion or other linear form, integration may be determined analytically by adding areas of trapezoids and triangles.

A simple assumption can be made on how the value of W_0 may affect the system's measurement to be controlled for simulation. For example, ΔT, the change in temperature T, may be assumed to be proportional or a linear function of W_0. Starting with an initial T, at each time step, we would have a new value of T as old $T + \Delta T$, compute new values of E, ΔE, and W_0. We may be able to see how temperature converges to the target temperature - a simulation of fuzzy control.

For commercial applications, the principles of fuzzy control discussed here may not actually operate inside individual machines, since fuzzy control is too costly and time consuming for realtime control. Instead, input to output mappings are determined using fuzzy control at the factories, these mappings are recorded on a computer chip, and the machines operate based on this information.

5.5.3 Extended Fuzzy if-then Rules Tables

Note on the four "corner regions" of fuzzy if-then rule tables

Up to this point, all the entries of the four corner regions of fuzzy if-then tables are assumed 0 for simple implementations. For more fine-tuned control, these corner regions can have nonzero values as in the following example.

W		NB	NM	NS	ΔE ZO	PS	PM	PB
	NB				PB	PM		
	NM				PM			
	NS				PS	ZO		NM
E	ZO	PB	PM	PS	ZO	NS	NM	NB
	PS	PM		ZO	NS			
	PM				NM			
	PB			NM	NB			

In this improved table, six nonzero entries are added in the two corner regions. Note that the same entries (e.g., *PM*) appear along the diagonal lines. In this sub-section, we discuss the significance of the four-corner regions.

1. For simple implementation discussed earlier, we assume $W = 0$ for all entries in all the four corner regions (here we call the original table). The justifications are as follows:

   i) The regions correspond to less critical points in control process.
   ii) The table with zero corner regions still covers a wide range (e. g., *ZO* = -0.33 to 0.33). Since multiple rules are used, even if $W = 0$ for some rules, $W \neq 0$ for some other rules and final $W \neq 0$.
   iii) W is typically, e. g., a *changing rate* of a heat source, rather than heat source itself. $W = 0$ means to keep the current heat source, and this works well for many applications. Incidentally, if W represents heat source itself, we cannot set $W = 0$; it probably means disastrous control.

2. An improved table over $W = 0$.
   The literature (e.g., Lee, p. 413) suggests adding six entries (e.g., if $E = NB$ and $\Delta E = PS$ then $W = PM$; other corner entries remain 0) for possible finer control as shown in the above table (an improved table). This is based on considering fine tuning in system response as explained below.

   In the following, Points refer to the points in Fig. 5.20.

   Point *a*: $E = NB$, $\Delta E = 0 \rightarrow W = PB$ (original table).
   Point *b*: $E = NB$, $\Delta E = PS$. To accelerate temperature increase, we may set $W = PM$ instead of 0.
   Point *c*: Typically, there is some time lag between fuzzy control and physical target systems. The added $W = PM$ at Point b may cause over-shooting. To correct this situation, we may start an early adjustment before Point *d* by: $E = NS$, $\Delta E = PB \rightarrow W = NM$ instead of 0.
   Point *d*: $E = ZO$, $\Delta E = PB \rightarrow W = NB$ (original table)

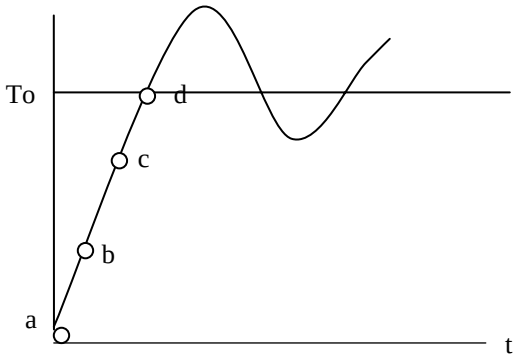


Fig. 5.20. System response points for finer fuzzy control.

The resulting typical system response may look as follows.

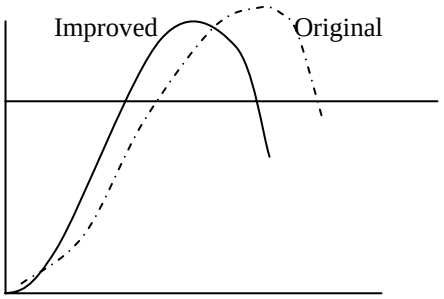


Fig. 5.21. System response with improved fuzzy control.

3. Further extensions for even finer tuning. Fill in the four corner regions with nonzero entries diagonally. For seven variables, a total of $7 \times 7 = 49$ entries:

		ΔE						
	W	*NB*	*NM*	*NS*	*ZO*	*PS*	*PM*	*PB*
	NB	*PB*	*PB*	*PB*	*PB*	*PM*	*PS*	*ZO*
	NM	*PB*	*PB*	*PB*	*PM*	*PS*	*ZO*	*NS*
	NS	*PB*	*PB*	*PM*	*PS*	*ZO*	*NS*	*NM*
E	*ZO*	*PB*	*PM*	*PS*	*ZO*	*NS*	*NM*	*NB*
	PS	*PM*	*PS*	*ZO*	*NS*	*NM*	*NB*	*NB*
	PM	*PS*	*ZO*	*NS*	*NM*	*NB*	*NB*	*NB*
	PB	*ZO*	*NS*	*NM*	*NB*	*NB*	*NB*	*NB*

For five fuzzy variables, 5 × 5 = 25 entries.

				ΔE		
W		*NB*	*NS*	*ZO*	*PS*	*PB*
	NB	*PB*	*PB*	*PM*	*PS*	*ZO*
	NS	*PB*	*PB*	*PS*	*ZO*	*NS*
E	*ZO*	*PB*	*PS*	*ZO*	*NS*	*NB*
	PS	*PM*	*ZO*	*NS*	*NB*	*NB*
	PB	*ZO*	*NS*	*NB*	*NB*	*NB*

For example, for $E = PB$, $\Delta E = NB$, *PB* and *NB* cancel out and $W = ZO$.

An extension of the case study for which both *E* and ΔE are nonzero

Example. $E = 3$ and $\Delta E = -0.5$, rather than $E = 3$ and $\Delta E = 0$.

The basic principle is the same as discussed in the cookbook recipe, p. 149.

Step 1. Each of E and ΔE are represented by two fuzzy variables: E is the same as before, i. e., *PB* with 0.5 and *PS* with 0.5. ΔE is now *ZO* with 0.75 and *NS* with 0.25.

Step 2. (a) The fuzzy if-then rule table has to be extended to have additional entries: (tentatively called Rules 10 and 11):

Rule 10. if $E = PS$ and $\Delta E = NS$ then $W = ZO$.
Rule 11. if $E = PB$ and $\Delta E = NS$ then $W = NS$.

The rest of the algorithm is performed in the same way.

(b) Compute the firing strengths, α_8, α_9, α_{10}, α_{11}.
(c) Find the membership function for W associated with each rule, $m_8(W)$, . . ., $m_{11}(W)$.
(d) Determine $m_T(W)$ by the max operations on $m_8(W)$, . . ., $m_{11}(W)$.

Step3. Compute the center of gravity.

5.5.4 A Note on Fuzzy Control Expert Systems

There are many control problems in which rules are expressed in descriptive rather than numeric expressions. For example, "if the speed is moderately fast, then slightly reduce the fuel amount" is a descriptive expression, while "if the speed is 200 km/hour, then reduce the fuel amount by 6%" is a numeric expression. Many real-world control rules are described in descriptive expressions, because this is the way experts perform their operations. Numeric rules are not used for several reasons.

For example, the number of rules required in numeric form may be large, the rules may be so complex that some sort of approximations have to be used (e.g., linearization of nonlinear expressions), and so forth. Even if we have numeric rules, obtaining numeric input can be either difficult or not economical. For these situations, descriptive rules with either numeric or descriptive input can be used.

For example, consider parallel parking a car. Solving this problem by, say, using a set of differential equations and measuring the distance, angular velocity of the steering, etc., is difficult. Instead, we may have been using a sort of a set of fuzzy rules, such as "if the distance to the next car is *PS* (Positive Small), then keep the same speed and rotate the steering *PS*."

As we have seen in this chapter, fuzzy control can easily incorporate descriptive rules in the system. For example, "if the speed is moderately fast (*PS*) and the angle is sharp to the left (*NM*: Negative Medium), then slightly reduce (*NS*) the fuel amount" can be a fuzzy rule described by an experienced expert. Membership functions can be defined for these fuzzy variables, fuzzy inference can be performed, and output can be computed. In a sense, this type of fuzzy control *quantifies* descriptive rules for numeric computation. This is the basic idea of implementing human operators' descriptive knowledge in the form of fuzzy control. Fuzzy control based on this idea has been most successful in terms of real-world applications among fuzzy systems, and the number is likely to increase in the future.

5.6 Hybrid Systems

One of the most active recent trends is the use of various forms of hybrid systems combining fuzzy logic and other areas such as neural networks and genetic algorithms.

Fuzzy - neural network hybrid systems

Much current research suggests using fuzzy logic and neural networks as complementary techniques. The fundamental concept of such hybrid systems is to complement each other's weaknesses, thus creating new problem-solving approaches. For example, there are no capabilities of machine learning in fuzzy systems. Nor do fuzzy systems have capabilities of memory or pattern recognition in the way neural networks do. The backpropagation neural network model, for instance, can memorize and recognize a potentially huge number of input patterns by storing much less information about weights. (For example, if there are 100 input units, and each unit can have either 0 or 1, then $2^{100} \approx 10^{30}$ different input patterns are possible.) Fuzzy systems with neural networks may add such capabilities, and in fact, recent commercial applications of neural networks in Japan are mostly tied to fuzzy control. The current stage of such neural network systems is relatively simple for real-world applications, however, and some people say that their functions are mostly "tuning" rather than "learning." Several applications have shown the advantages of neural networks in mapping the non-linear behavior of systems to predict future states, monitor the system behavior and anticipate failures. (For more, see Jang, et.al., 1997.)

Fuzzy - genetic algorithm hybrid systems

Applications of genetic algorithms combined with fuzzy control are being investigated not only at the academic level but also at the commercial level. Genetic algorithms are particularly well-suited for tuning the membership functions in terms of placing them in the universe of discourse. Properly configured genetic algorithm/fuzzy architecture searches the complete universe of discourse and finds adequate solutions according to the fitness function.

Fuzzy - PID hybrid systems

For certain applications, fuzzy and PID systems are employed together as a hybrid controller. A PID controller can be used for approximate and fast control, while a fuzzy system either tunes the PID gains or schedules the most appropriate PID controller for better performance.

5.7 Fundamental Issues

Problems and limitations of fuzzy systems

1) Stability
   Stability is a major issue for fuzzy control. As described below, there is no theoretical guarantee that a general fuzzy system will not become chaotic and stays stable, although such a possibility appears to be extremely slim from the extensive experience.

2) Lack of learning capability
   As mentioned before, fuzzy systems lack capabilities of machine learning, and neural network-like memory and pattern recognition. This is why hybrid systems, particularly neuro-fuzzy systems, are becoming popular for certain applications.

3) Determining or tuning good membership functions and fuzzy rules is not always easy. Even after extensive testing, it is difficult to say how many membership functions are really required. Questions like why a particular fuzzy expert system needs so many rules, or when a developer can stop adding more rules are not easily answered.

4) There exists a general misconception of the term "fuzzy" as imprecise or imperfect. Many professionals think of fuzzy logic as "magical" without firm mathematical foundation.

5) Verification and validation of a fuzzy expert system generally requires extensive testing with hardware in the loop. Such a luxury may not be affordable by all developers.

Stability, controllability and observability

The notion of stability is well-established in the classical control theory, and for a given linear system, several criteria of stability can be applied and necessary computations can be performed to obtain results. Similarly, the notion of controllability and observability is firmly established in modern state space theory. Using a linearized set of equations, proper parameters can be computed to show that the system behavior meets these criteria well. As a result of the complexity of mathematical analysis for fuzzy logic, stability theory requires further study, and issues such as controllability and observability have to be defined for fuzzy control systems.

Fuzzy versus probability theories

The continuous rather than crisp transition characteristics between 0 and 1 in fuzzy sets and logic is similar to probability theory. Additionally, the technique of deriving membership functions using the relative frequency distribution confuses developers and creates an impression that fuzzy logic is another form of probability theory. This sometimes raises debates about how fuzzy theory differs from probability.

The most fundamental difference is in their basic ideas. In probability theory, we deal with chance of occurrence, e.g., getting the head by flipping a coin, winning a lottery, being involved in a car accident, and so forth. The membership degree of fuzzy set theory is not probability, but plausibility. For example, suppose that someone's membership degree in a set of young people is 0.7. This does not mean that this person is young 70% of time and old the remaining 30% of the time, which reflects the probability. Rather, it means that this person is fairly young to the degree of 70% right now all the time. In other words, the fundamental difference between fuzzy and probability theories is that the former deals with deterministic plausibility, while the latter deals with the likelihood of nondeterministic, stochastic events.

From a practical point of view, fuzzy systems have led to numerous new real world applications, which would not have been realized by using probability theory.

5.8 Additional Remarks

A bit of history

Fuzzy set theory was introduced in 1965 by Lotfi A. Zadeh at the University of California at Berkeley. In 1974, E.H. Mamdani et. al at the University of London, demonstrated applications of fuzzy set theory to control problems. But the concepts were known in a relatively small research community until Hitachi of Japan used fuzzy control for the new Sendai Subway in 1986. The performance improvement was significant; it reduced the stop-gap distance by 2.5 times, doubled the comfort index, and saved 10% in power consumption. The number of practical fuzzy application systems has exploded since then. World-wide industrial and commercial applications appear likely to increase significantly in the near future. Academic

interests in fuzzy theory has also been growing recently, as indicated by the first IEEE Transactions on Fuzzy Systems in February 1993.

Significance of fuzzy control

As stated at the beginning of this chapter, fuzzy logic allows decision making under fuzzy information and rules. It also allows us to represent descriptive or qualitative expressions. For control problems that involve the fuzziness and descriptive expressions, fuzzy control is typically simpler, faster, less costly and more robust than traditional mathematical approaches. Fig. 5.22 shows the approximate position for which fuzzy control is most useful. In many systems (e.g., "humanistic" systems), using classical control makes precision either impossible or inappropriate. Here are some comparisons for typical situations.

Typical classic versus fuzzy control systems

	Classic Control	Fuzzy Control
Input, output, and intermediate values	Numeric	Numeric+ descriptive
Algorithm	Single, e.g., a differential equation	Multiple (Front-end if-then rules may select algorithms)
Robustness	Weak	Good

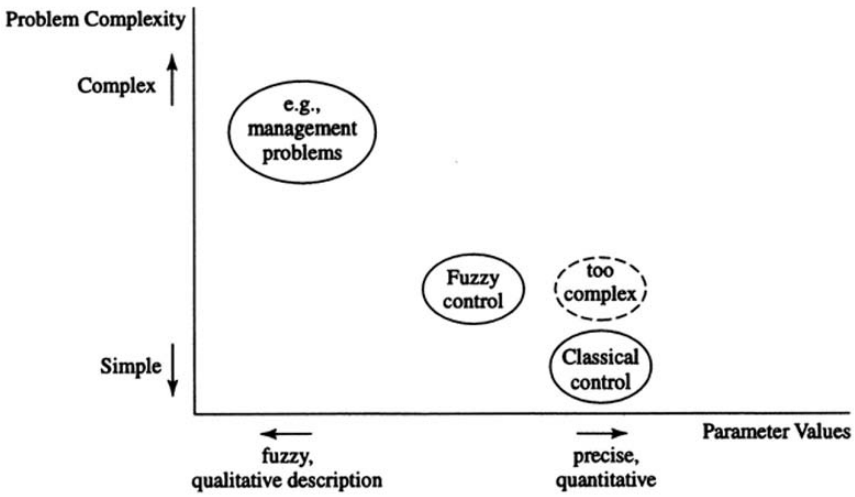


Fig. 5.22. Approximate domain for which fuzzy control best fits.

Generic categories of fuzzy system applications

The following is a list of different categories and their fuzzy system application types.

Category	Application Area Examples
Control	Control is the most widely applied category today. The majority of the industrial applications in the next table are in this category.
Pattern recognition	Image (e.g., optical character recognition) audio, signal processing.
Quantitative analysis	Operations research, statistics, management
Inference	Expert systems for diagnosis, planning, and prediction; natural language processing; intelligent interfaces; intelligent robots; software engineering.
Information retrieval	Databases.

A partial list of application areas of fuzzy systems

The following is a list of selected application areas and examples.

Field	Applications
Transportation	Subways, helicopters, elevators, traffic control, highway tunnel-air control
Automobiles	Transmissions, cruise control, engines, brakes
Consumer electronics	Washing machines, driers, refrigerators, vacuum cleaners, rice cookers, televisions, VCRs, air conditioners, kerosene fan heaters, microwave ovens, shower systems, video cameras
Robotics	
Computers	
Other industries	Steel, chemical, power generation, construction, nuclear, aerospace
Engineering	Electrical, mechanical, civil, environmental, geophysics
Medicine	
Management	Credit evaluation, damage/risk assessment, stock picking, marketing analysis, production management, scheduling, decision support systems

Further Reading

If I had to, I would choose Terano's book for general introduction to fuzzy systems and applications. Lee's article is a clear tutorial on fuzzy control. Zadeh's 1965 piece is the seminal article on fuzzy sets and further derivatives of many areas of fuzzy systems.

J.-S.R. Jang, C.T. Sun and E. Mizutani, *Neuro-Fuzzy and Soft Computing*, Prentice-Hall, Upper Saddle River, NJ, 1997.

Y. Jin, *Advanced Fuzzy Systems Design and Applications*, Physica-Verlag, 2003.

C.C. Lee, "Fuzzy Logic in Control Systems: Fuzzy Logic Controller, Parts I and II." *IEEE Transactions on Systems, Man and Cybernetics*, 20, 2 (March/April, 1990) 404-435.

T. Munakata and Y. Jani, "Fuzzy Systems: An Overview," *Communications of the ACM*, Vol. 37, No. 3 (March, 1994), 69-76.

T. Terano, K. Asai, and M. Sugeno, *Fuzzy Systems Theory and Its Applications*, Academic Press, San Diego, 1992.

P. Witold and G. Fernando, *An Introduction to Fuzzy Sets: Analysis and Design*, MIT Press, 1998.

L.A. Zadeh, "Fuzzy Set," *Information and Control*, Vol. 8, 1965, 338-353.

L.A. Zadeh, "Fuzzy Algorithms," *Information and Control*, Vol. 12, 1968, 94-102.

L.A. Zadeh, "Outline of a New Approach to the Analysis of Complex Systems and Decision-Making Approach," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SME-3, No. 1, January, 1973, 28-44.

H. -J Zimmermann and Hans-Jurgen Zimmerman, *Fuzzy Set Theory - and Its Applications*, 4th Ed., Springer, 2005.

Journals

IEEE Transactions on Fuzzy Systems.

Fuzzy Sets and Systems, Elsevier (sponsored by the IFSA, International Fuzzy Systems Association).

International Journal of Approximate Reasoning, Elsevier (affiliated with the NAFIPS, North American Fuzzy Information Processing Society).

Many other journals, magazines and conference proceedings in AI and its applications carry articles in fuzzy systems.

6 Rough Sets

6.1 Introduction

Rough set theory is a relatively new mathematical and AI technique introduced by Zdzislaw Pawlak, Warsaw University of Technology, in the early 1980s. This area has remained unknown to most of the computing community until recently. Rough set theory is particularly useful for discovering relationships in data. This process is commonly called knowledge discovery or **data mining**. It is also suited to reasoning about imprecise or incomplete data.

Rough sets, meaning approximation sets, are built on ordinary sets. We recall that fuzzy sets are a generalization of ordinary sets. In this regard, rough sets and fuzzy sets have a common ground. However, their ways of deviating from ordinary sets are different, and their primary application objectives are also different. These two approaches can be used in their own domains independently, or they can be complementary. Briefly, fuzzy sets allow partial membership to deal with gradual changes or uncertainties. Rough sets, on the other hand, allow multiple memberships to deal with indiscernibility.

Rough set theory is commonly compared to other techniques such as statistical analysis, particularly discriminant analysis, and machine learning in classical AI. While there are no mathematical proofs to show which technique is most suitable for what types of problems, it appears that each technique has specific strengths for problems of certain kinds. For example, rough sets might do a better job than statistical analysis when the underlying data distribution deviates significantly from a normal distribution, since there is no such distribution assumption in rough sets. Also, perhaps rough sets could be better than statistical analysis when the sample size is small, since any distribution can hardly be defined in such a case. Machine learning can broadly be defined as the process in which computers acquire their knowledge and improve their skills by themselves. Under this broad definition, the data mining aspect of rough sets can be regarded as a technique of machine learning. Rough sets, however, employ an approach different from those found in classical machine learning.

A major feature of rough set theory in terms of practical applications is the classification of empirical data and subsequent decision making. The primary application domains of rough sets have been in symbolic approaches such as data and

decision analysis, databases, knowledge based systems, and machine learning. There are also recent interests in rough control, applications of rough set theory to control problems. Practical application areas include: engineering disciplines such as civil, electrical, chemical, mechanical and transportation; pharmacology; medicine; and operations research. Specific application examples include: cement kiln, aircraft pilot performance evaluation, hydrology, and switching circuits.

The basic idea of rough sets

Raw data is often very detailed, yet disorganized, incomplete and imprecise. To understand and use the data, we derive underlying knowledge about the data, i.e., what it represents. Such knowledge can be represented in many forms. Rules are the most common form of representing knowledge. Other forms include equations and algorithms.

In many situations, we may not need detailed data to derive conclusions for actions. Instead, "coarse" or "rough" data or data sets may be sufficient. In certain situations, such approximate rough data may be even better than a detailed one. Too much detail is often confusing. Rough data can be more efficient, effective and robust, and may uncover the underlying characteristics.

The rough sets methodology gives a new technique for reasoning from imprecise and ambiguous data. The technique can efficiently perform knowledge acquisition and machine learning. It performs these by lowering the degree of precision in data, based on a rigorous mathematical theory. By selecting the right roughness or precision of data, we will find the underlying characteristics of data. In this chapter we will give a brief introduction to rough set theory. More short introductions will be found in Pawlak 1988 and 1995. A thorough treatment of the theory, especially its theoretical foundation, is discussed in Pawlak 1991.

List of selected symbols in this chapter

Symbol	Page	Meaning
$U \times V$	165	The cartesian product of two sets, U and V.
R^*	167, 169	The partition induced by an equivalence relation R: $R^* = \{X_1, X_2, ..., X_n\}$, where X_i is an equivalence class of R. X_i is also called an elementary set of an approximation space $S = (U, R)$.
$R^*_1 \cdot R^*_2$	167	The product of two partitions, R^*_1 and R^*_2, is the partition induced by $R_1 \cap R_2$.
$S = (U, R)$	171	An approximation space, where U is a finite set of objects and $R \subseteq U \times U$ is an equivalence relation on U. If $u, v \in U$ and $(u, v) \in R$, we say that u and v are indistinguishable in S. R is called an indiscernibility relation.

Symbol	Page	Meaning
$\underline{S}(X)$	171	The lower approximation of X in $S = \cup_{Xi \subseteq X} X_i$
$\overline{S}(X)$	171	The upper approximation of X in $S = \cup_{Xi \cap X \neq \varnothing} X_i$
$POS_S(X)$	172	The positive region of X in $S = \underline{S}(X)$
$BND_S(X)$	172	The boundary region of X in $S = \overline{S}(X) - \underline{S}(X)$
$NEG_S(X)$	172	The negative region of X in $S = U - \overline{S}(X)$
a	174	The confidence factor
K	176	A knowledge representation system, where: $K = (U, C, D, V, \rho)$, U is a set of objects; C is a set of condition attributes; D is a set of action (or decision) attributes; $V = \cup_{a \in F} V_a$, and V_a is the domain of attribute $a \in F$, where $F = C \cup D$; ρ: $U \times F \rightarrow V$ for every $u \in U$ and $a \in F$ is an information function. ρ can also be represented as: ρ_u: $F \rightarrow V$ by $\rho_u(a) = \rho(u, a)$ for every $u \in U$ and $a \in F$.
$\tilde{G}$	180	An equivalence relation defined on U for any subset G of C or D, such that $(u_i, u_j) \in \tilde{G}$ if and only if $\rho(u_i, g) = \rho(u_j, g)$ for every $g \in G$.
$POS_S(B^*)$	183	The positive region of partition B^* in $S = \cup_{Yj \in B^*} \underline{S}(Y_j) = \cup_{Yj \in B^*} [\cup_{Xi \subseteq Yj} X_i]$.
$BND_S(B^*)$	183	Boundary region $= \cup_{Yj \in B^*} (\overline{S}(Y_j) - \underline{S}(Y_j)) = \cup_{Yj \in B^*} [\cup_{Xi \cap Yj \neq \varnothing} X_i - \cup_{Xi \subseteq Yj} X_i]$
$NEG_S(B^*)$	183	Negative region $= U - \cup_{Yj \in B^*} (\overline{S}(Y_j)) = U - [\cup_{Xi \cap Yj \neq \varnothing} X_i]$.
$\gamma_A(B)$	184	The dependency of B on $A = \lvert POS_S(B^*) \rvert / \lvert U \rvert$, where $\lvert\ \rvert$ denotes the cardinality, and $A \subseteq C$, the set of conditional attributes and $B \subseteq D$, the set of decision attributes
$A \rightarrow_\gamma B$	184	The dependency of B on A, where A, B and γ as defined above
$\beta_A(B)$	185	The discriminant index of B on $A = \lvert POS_S(B^*) \cup NEG_S(B^*) \rvert / \lvert U \rvert$
$\sigma_A(B)$	185	The significance of B on A, defined as $\sigma_A(B) = \gamma_C(B) - \gamma_{C\text{-}A}(B)$

Symbol	Page	Meaning
$\hat{C}$	186	A reduct or relative reduct of C. (A subset B of C is independent if there is no other subset B' of C which is $B' \subset B$ and $\tilde{B}' = \tilde{B}$. B is a reduct of C if B is a maximal independent subset. This is extended to a relative reduct by considering subset B to be an independent set with respect to D, where there is no other subset B' that satisfy $\mathrm{POS}_{B'}(D^*) = \mathrm{POS}_B(D^*)$.)
$\mathrm{RED}(C)$	186	The collection of all reducts of C
$\mathrm{RED}_D(C)$	186	The collection of all relative reducts of C with D
$\mathrm{CORE}(C)$	187	Core of $C = \cap_{B \in \mathrm{RED}(C)} B$
$\mathrm{CORE}_D(C)$	188	Relative core of $C = \cap_{B \in \mathrm{RED}D(C)} B$

6.2 Review of Ordinary Sets and Relations

For the convenience of the reader, we will briefly review some basics on sets and relations which will be used in this chapter. A review on the topics with a different focus is given in Subsection 5.3.1. These topics are typically covered in college discrete mathematics courses and a reader who is familiar with these materials may skip this section. (A reader who needs more details may see, e.g., C.L. Liu, *Elements of Discrete Mathematics*, 2nd Ed., McGraw-Hill, 1985, or similar discrete mathematics books.)

Given two sets, U and V, we define the **cartesian product** = $U \times V$ as $U \times V = \{(u, v) \mid u \in U, v \in V\}$, where (u, v) represents an ordered pair. That is, $U \times V$ is the set of all such ordered pair elements where u is chosen for every element of U and v is chosen for every element of V. For example, suppose that U is the set of neckties a man has: U = {red-tie, blue-tie}, and V is the set of shirts he owns: V = {white-shirt, grey-shirt, pink-shirt}. Let us call them red, blue, white, grey and pink for simplicity. $U \times V$ in this example then is {(red, white), (red, grey), (red, pink), (blue, white), (blue, grey), (blue, pink)}. In general, if U has m elements and V has n elements, then $U \times V$ has $m \times n$ elements. A **binary relation**, or simply **relation**, R from U to V is a subset of $U \times V$. In the above example, {(red, white), (red, grey), (blue, white), (blue, pink)} is a binary relation. Fig. 6.1 shows these concepts. Especially, if $U = V$, the cartesian product becomes $U \times U$. A binary relation is a subset of $U \times U$ and is said to be a relation on U (Fig. 6.2).

We can define various kinds of relations on U when they satisfy specific characteristics. In particular, R is an **equivalence relation**, if

(1) reflexive, i.e., (u, u) for every $u \in U$.
(2) symmetric, i.e., (u, v) implies (v, u) for every $u, v \in U$.
(3) transitive, i.e., (u, v) and (v, w) imply (u, w) for every $u, v, w \in U$.

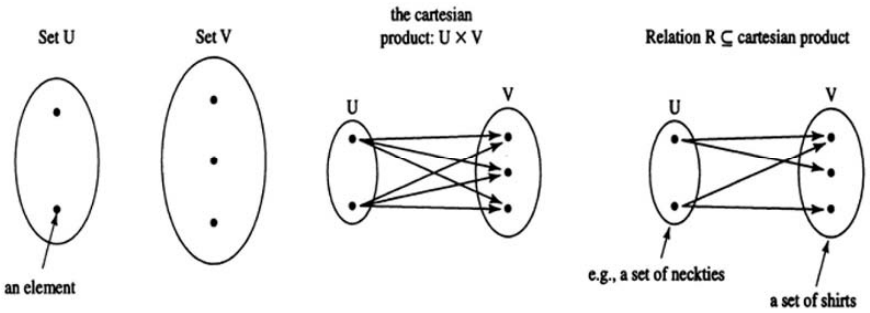


Fig. 6.1. The cartesian product and a binary relation defined on two sets.

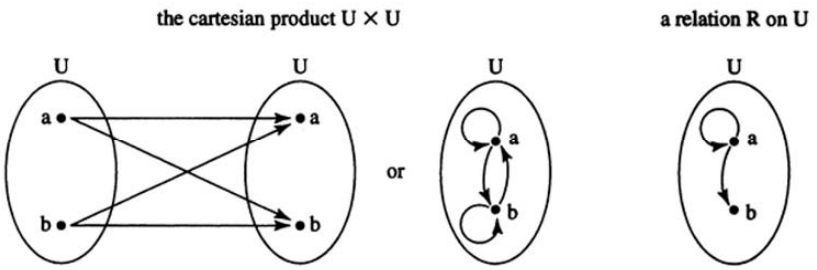


Fig. 6.2. The cartesian product and a binary relation defined on one set.

Example 1.

Let NY = New York, LA = Los Angeles, SF = San Francisco, MO = Montreal, TO = Toronto, MC = Mexico City, and U = {NY, LA, SF, MO, TO, MC}. Then,

R ={(u, v) | u and v are in the same country} = {(NY, NY), (LA, LA), (SF, SF), (MO, MO), (TO, TO), (MC, MC), (NY, LA), (LA, NY), (NY, SF), (SF, NY), (LA, SF), (SF, LA), (MO, TO), (TO, MO)} is an equivalence relation.

Example 2.

U = a group of people.

R_1 = {(u, v) | u and v have the same last name} is an equivalence relation.

R_2 = {(u, v) | u and v have the same birthday} is an equivalence relation.

R_3 = {(u, v) | u and v have the same sex} is an equivalence relation.

R_4 = $R_1 \cap R_2$ = {(u, v) | u and v have the same last name and the same birthday} is an equivalence relation.

R_5 = $R_1 \cap R_2 \cap R_3$ = {(u, v) | u and v have the same last name, the same birthday, and the same sex} is an equivalence relation.

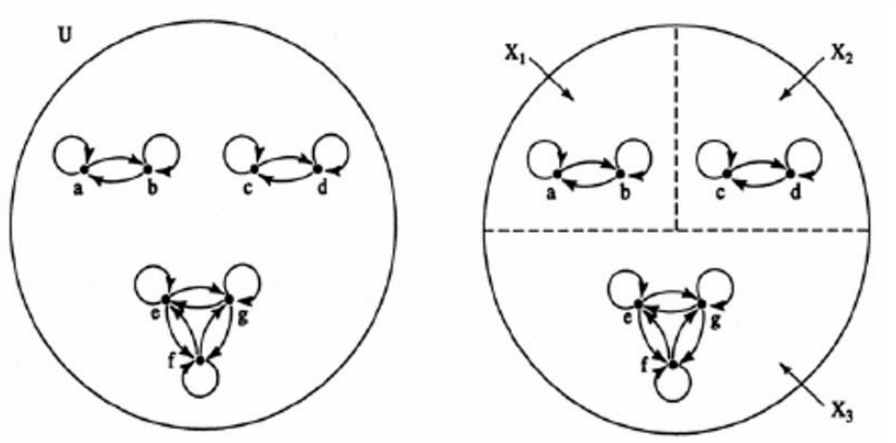
Example 3.

U = {a, b, c, d, e, f, g}, R = {(a, a), (b, b), (c, c), (d, d), (e, e), (f, f), (g, g), (a, b), $(b,$

a), (*c*, *d*), (*d*, *c*), (*e*, *f*), (*f*, *e*), (*e*, *g*), (*g*, *e*), (*f*, *g*), (*g*, *f*)} is an equivalence relation. As in case for any mathematical treatment, such representation can be an abstract form of specific cases such as Examples 1 and 2. Fig. 6.3(a) is a diagram representation of this relation.

Generally, a **partition** of a set U is a set of nonempty subsets of U, $\{X_1, X_2,..., X_k\}$, where $X_1 \cup X_2 \cup ... \cup X_k = U$ and $X_i \cap X_j = \varnothing$ for $i \neq j$. i.e., a partition divides a set into a collection of disjoint subsets. These subsets are called **blocks** of the partition. In particular, when we have an equivalence relation R on U, we can partition U so that every two elements in each subset are related and any two elements in different subsets are unrelated. We say the partition is **induced** by the equivalence relation R, and denote the partition as R^*. The subsets are called the **equivalence classes**. Fig. 6.3(a) is an example of an equivalence relation, and Fig. 6.3(b) is the partition induced by the equivalence relation. Sets X_1, X_2, and X_3 are equivalent classes.

Let R_1 and R_2 be two equivalence relations on U, and R^*_1 and R^*_2 be the corresponding partitions induced by R_1 and R_2. The **product of two partitions**, R^*_1 and R^*_2, denoted $R^*_1 \cdot R^*_2$ is defined as the partition induced by $R_1 \cap R_2$. In other words, in the partition $R^*_1 \cdot R^*_2$, two elements a and b are in the same block (equivalent class) if a and b are in the same block of R^*_1 *and* also in the same block of R^*_2. In Example 2, in the partition induced by $R_4 = R_1 \cap R_2$, two persons will be in the same block if they have the same last name and the same birthday.



(a) (b)

Fig. 6.3 (a) An equivalence relation. (b) The partition induced by the equivalence relation.

6.3 Information Tables and Attributes

Rough set theory deals with data expressed in two-dimensional or matrix form tables, called information tables. In this section, we will discuss some terminology associated with information tables.

Information tables

For rough set theory, the input information is given in the form of a two-dimensional table (i.e., matrix), called an **information table** (or **decision table**). The following is a simple example of an information table.

Example 4.

Table 1. Symptoms and heart problems of patients

Universe	Condition Attributes		Decision Attribute
Person	Temperature	Blood Pressure	Heart Problem
Adams	normal	low	no
Brown	normal	low	no
Carter	normal	medium	yes
Davis	high	medium	no
Evans	high	high	yes
Ford	high	high	yes

As we see in this example, the columns of an information table are divided into three sections; the **universe** *U*, **condition attributes** (or simply **attributes**), and **decision attributes** (or simply **decisions**). The universe is the set of elements under consideration as in ordinary sets. In the above example, the universe contains 6 patients. We can have any number of condition attributes; in this example, we have two, Temperature and Blood Pressure. Similarly, we can have any number of decision attributes, although one is all that is required. In the above example, we have one decision attribute, Heart Problem. If we want, we can add more, as for example, Stroke Problem, Diabetes Problem, and so on. Rows of an information table are called **entities** (**objects**, or sometimes **examples**). The entities can be labeled by the elements of the universe as, for example, Adams, etc. Each element (patient in the above table) is characterized by its condition and decision attribute values.

The previous table is simple for an illustration purpose. For practical applications, the table size would be much larger. For example, there may be 10,000 patients, twenty condition attributes, and the range of each attribute may be much higher than simply "normal" and "high." Also, the application domains can be in many other areas such as analysis of consumer and industrial products, process control, and so on. Given input in this form, we would like to derive various conclusions as our output. Possible types of conclusions include: how the decision attributes depend on the condition attributes, are there any condition attributes that are redundant, i.e., can be eliminated without affecting the decision making, and derivation of underlying rules governing the relationship from the condition to decision attributes.

In the above example, let Adams = *a*, Brown = *b*, Carter = *c*, Davis = *d*, Evans = *e*, and Ford = *f*, for simplicity. Then the universe *U* is {*a*, *b*, *c*, *d*, *e*, *f*}. Equivalence relations can be defined by the condition and decision attributes, as for example,

$R_1 = \{(u, v) \mid u \text{ and } v \text{ have the same Temperature}\}$
$R_2 = \{(u, v) \mid u \text{ and } v \text{ have the same Blood Pressure}\}$
$R_3 = \{(u, v) \mid u \text{ and } v \text{ have the same Heart Problem}\}$
$R_4 = R_1 \cap R_2 = \{(u, v) \mid u \text{ and } v \text{ have the same Temperature and Blood Pressure }\}$.

The universe can then be partitioned by these equivalence relations. For example, R_4*, the **partition** induced by the equivalence relation $R_4 = R_1 \cap R_2 = \{(u, v) \mid u$ and v have the same Temperature and Blood Pressure$\}$, is $R_4^* = R_1^* \cdot R_2^* = \{X_1, X_2, X_3, X_4\}$, where $X_1 = \{a, b\}$, $X_2 = \{c\}$, $X_3 = \{d\}$, $X_4 = \{e, f\}$. Sets X_1, X_2, X_3, and X_4 are called the **equivalence classes**.

Concepts

So far we have focused our attention primarily on condition attributes. For decision attributes, we can define equivalence relations and determine the partitions in the same way as for condition attributes. For example, we can define the following equivalence relation.

$$R_3 = \{(u, v) \mid u \text{ and } v \text{ have the same Heart Problem}\}$$

The partition induced by this relation is:

$$R_3^* = \{Y_1, Y_2\}, \text{ where } Y_1 = \{a, b, d\} \text{ and } Y_2 = \{c, e, f\}.$$

In general, such sets in a partition are called **concepts** (Y_1 and Y_2 in the above example). The concept Y_1 corresponds to the set of all patients with no heart problem, and Y_2 with heart problem.

In rough set theory, we are interested in finding mappings from the partitions induced by the condition attributes to the partitions induced by decision attributes.

Rule induction

We saw that R_4*, the partition induced by the equivalence relation $R_4 = R_1 \cap R_2 = \{(u, v) \mid u$ and v have the same Temperature and Blood Pressure$\}$, is:

$$R_4^* = \{X_1, X_2, X_3, X_4\}, \text{ where } X_1 = \{a, b\}, X_2 = \{c\}, X_3 = \{d\}, X_4 = \{e, f\}.$$

In the above, we had R_3*, the partition induced by $R_3 = \{(u, v) \mid u$ and v have the same Heart Problem$\}$ as:

$$R_3^* = \{Y_1, Y_2\}, \text{ where } Y_1 = \{a, b, d\} \text{ and } Y_2 = \{c, e, f\}.$$

From these, we can derive rules as follows:

if $X_1 = \{a, b\}$, then $Y_1 = \{a, b, d\}$.
if $X_2 = \{c\}$, then $Y_2 = \{c, e, f\}$.
if $X_3 = \{d\}$, then $Y_1 = \{a, b, d\}$.
if $X_4 = \{e, f\}$, then $Y_2 = \{c, e, f\}$.
In words,

if Temperature is normal and Blood Pressure is low, then no Heart Problem.
if Temperature is normal and Blood Pressure is medium, then yes Heart Problem.
if Temperature is high and Blood Pressure is medium, then no Heart Problem.
if Temperature is high and Blood Pressure is high, then yes Heart Problem.

Or, these rules can be simplified as follows:

if Blood Pressure is low, then no Heart Problem.
if Temperature is normal and Blood Pressure is medium, then yes Heart Problem.
if Temperature is high and Blood Pressure is medium, then no Heart Problem.
if Blood Pressure is high, then yes Heart Problem.

6.4 Approximation Spaces

In the previous Table 1, information processing is straightforward, since each elementary set (equivalence class) in the partition induced by the two condition attributes maps to an elementary set (a concept) in the partition induced by the decision attribute. In general, this may not be the case. That is, elements in an elementary set may map to different concepts. Dealing with such information tables is the core of rough set theory, and we will discuss some basics of these topics in this section.

Inconsistent information tables

Now consider Table 2, where a patient Gill is added to Table 1. Previously based on Table 1, we had a rule: "if Temperature is high and Blood Pressure is high, then yes Heart Problem." With the addition of Gill, this rule is no longer true. Such a table is called **inconsistent**. That is, an inconsistent information table contains entities whose condition attribute values are the same, but lead to different concepts.

TABLE 2. (Addition of a patient to Table 1)

Universe	Condition Attributes		Decision Attribute
Person	Temperature	Blood Pressure	Heart Problem
Adams	normal	low	no
Brown	normal	low	no
Carter	normal	medium	yes
Davis	high	medium	no
Evans	high	high	yes
Ford	high	high	yes
Gill	high	high	no

To deal with inconsistent tables, we introduce various terminology which is discussed in the following.

Approximation spaces and lower and upper approximations

As their name implies, rough sets are sets which cannot be clearly ascertained or defined. However, rough (approximate) sets can be constructed. We will define approximation spaces which lead to the concept of rough sets.

Let U be a finite set of objects and $R \subseteq U \times U$ be an equivalence relation on U. Then, $S = (U, R)$ is called an **approximation space**. If $u, v \in U$ and $(u, v) \in R$, we say that u and v are **indistinguishable** in S. R is called an **indiscernibility relation**. Indiscernibility relations are the main concept of rough sets. For example, in Table 2 above, suppose $R = R_2 = \{(u, v) \mid u$ and v have the same Blood Pressure$\}$. Then R_2 is a indiscernibility relation. Patient a (Adams) and b (Brown), for example, are indiscernible using this equivalence relation; so are elements c and d; so are e, f, and g. We can define other indiscernibility relations by choosing other equivalence relations, such as $R_4 = R_1 \cap R_2 = \{(u, v) \mid u$ and v have the same Temperature and the same Blood Pressure$\}$.

Let $R^* = \{X_1, X_2, ..., X_n\}$ denote the partition induced by R, where X_i is an equivalence class of R. X_i is also called an **elementary set** of S. Any finite union of elementary sets is called a **definable set**.

Let X be any subset of U. Then we define the following:

$\underline{S}(X) = \cup_{Xi \subseteq X} X_i$ the **lower approximation** of X in S

$\overline{S}(X) = \cup_{Xi \cap X \neq \varnothing} X_i$ the **upper approximation** of X in S

In words, $\underline{S}(X)$ is the union of all the elementary sets of S, where each elementary set is totally included (i.e., a subset) in X. $\overline{S}(X)$ is the union of all the elementary sets of S, where each elementary set contains at least one element in X. In the following, when the meaning is clear from the context we sometimes write $\underline{S}(X)$ as $\underline{S}$ and $\overline{S}(X)$ as $\overline{S}$ for simplicity.

Example 5.

In Table 2, let us choose R_2, an equivalence relation for the same Blood Pressure, as our relation R. Then $S = (U, R_2)$ is the approximation space. The partition induced by R_2 is $R_2^* = \{X_1, X_2, X_3\}$, where $X_1 = \{a, b\}$, $X_2 = \{c, d\}$, and $X_3 = \{e, f, g\}$ are the equivalence classes or elementary sets of S (Fig. 6.4a). $X_1 \cup X_2 = \{a, b, c, d\}$ is a definable set of S. Suppose that we have a new concept (for example, Stroke Problem = no) for which $X = \{b, c, d\}$. Then $\underline{S}(X) = X_2$ and $\overline{S}(X) = X_1 \cup X_2$ (Fig. 6.4b).

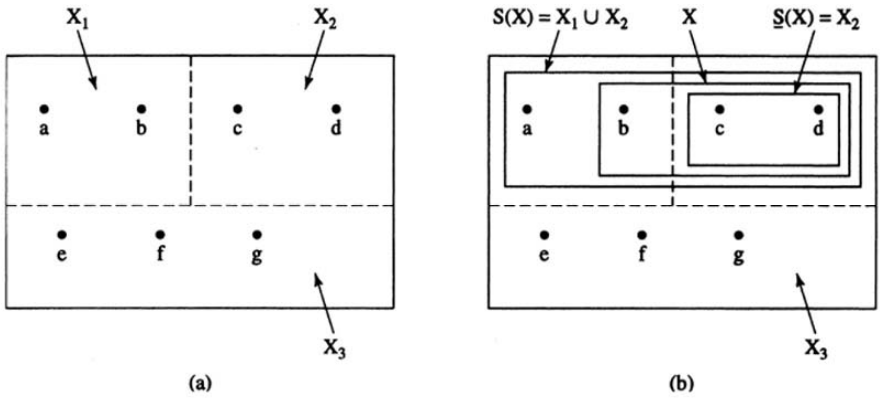


Fig. 6.4 (a) The partition induced by the equivalence relation of the same Blood Pressure. (b) The lower and upper approximation of $X = \{b, c, d\}$.

Three distinct regions in an approximation space: positive, boundary and negative

Using the lower and upper approximations discussed above, we can characterize the approximation space $S = (U, R)$ in terms of the concept X with three distinct regions defined as follows:

1. the **positive region**: $\text{POS}_S(X) = \underline{S}(X)$
2. the **boundary region**: $\text{BND}_S(X) = \overline{S}(X) - \underline{S}(X)$
3. the **negative region**: $\text{NEG}_S(X) = U - \overline{S}(X)$

The lower and upper approximations, and the positive, boundary, and negative regions are the most important notions in rough set theory.

In Table 2, suppose $X = \{b, c, d\}$, then the three distinct regions are (Fig. 6.5):

$$\text{POS}_S(X) = \underline{S}(X) = X_2$$
$$\text{BND}_S(X) = \overline{S}(X) - \underline{S}(X) = X_1 \cup X_2 - X_2 = X_1$$
$$\text{NEG}_S(X) = U - \overline{S}(X) = X_3$$

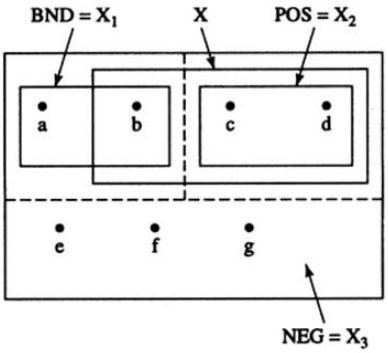
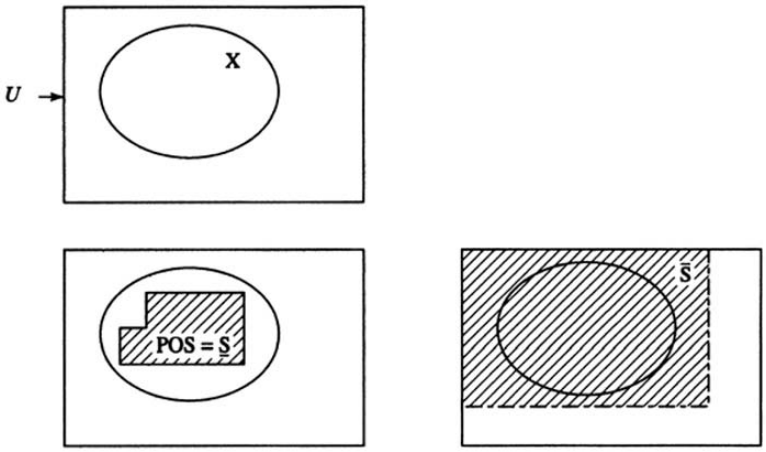


Fig. 6.5. The positive, boundary, and negative regions of $X = \{b, c, d\}$.

In general, a diagram interpretation of the three regions are given in the following Fig. 6.6. Suppose that the top three figures are given:



Then we have the following for the boundary and negative regions:

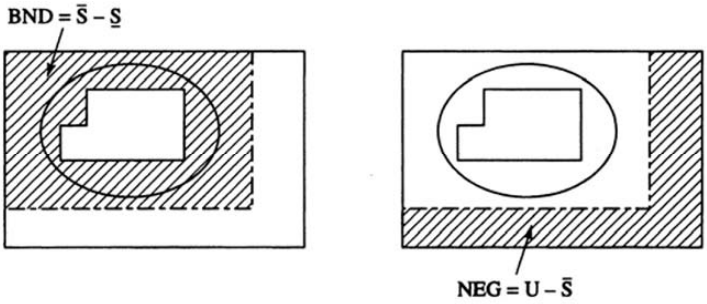


Fig. 6.6.A diagram interpretation of the lower and upper approximations and the positive, boundary, and negative regions.

Rule induction on an approximation space

For any concept, rules induced from its positive region (i.e., lower approximation) are called **certain**, since they are certainly valid. On the other hand, for any concept, rules induced from the boundary region of the concept are called **uncertain**. For an uncertain rule, we can define the **confidence factor** α. Let X_i be an elementary set in the boundary region and Y_j be a concept. The confidence factor for a rule derived from X_i and Y_j is:

$$\alpha = \frac{P\left|X_i \cap Y_j\right|}{\left|X_i\right|}$$

In words, the confidence factor is (the number of elements that are in the elementary set under consideration and that satisfy the concept for the rule) / (the total number of elements in the elementary set under consideration).

Example 6.

In Table 2, certain rules can be:

if $X_1 = \{a, b\}$, then $Y_1 = \{a, b, d, g\}$.
if $X_2 = \{c\}$, then $Y_2 = \{c, e, f\}$.
if $X_3 = \{d\}$, then $Y_1 = \{a, b, d, g\}$.

In words,

if Temperature is normal and Blood Pressure is low, then no Heart Problem.
if Temperature is normal and Blood Pressure is medium, then yes Heart Problem.
if Temperature is high and Blood Pressure is medium, then no Heart Problem.

Uncertain rules and their confidence factors can be:

if $X_4 = \{e, f, g\}$, then $Y_1 = \{a, b, d, g\}$ with $\alpha = |\{g\}| / |X_4| = 1/3 = 0.33$.
if $X_4 = \{e, f, g\}$, then $Y_2 = \{c, e, f\}$ with $\alpha = |\{e, f\}| / |X_4| = 2/3 = 0.67$.

In words,

if Temperature is high and Blood Pressure is high, then no Heart Problem with the confidence factor = 0.33.
if Temperature is high and Blood Pressure is high, then yes Heart Problem with the confidence factor = 0.67.

Definability and rough sets

As a special case, if BND = $\varnothing$, i.e., if $\underline{S}(X) = \overline{S}(X)$, X is a definable set in S (for example, the two concepts in Table 1 are definable sets). Otherwise, if BND $\neq \varnothing$, or

equivalently if $\underline{S}(X) \neq \overline{S}(X)$, then X is said to be **undefinable** or a ***rough set***. (The two concepts in Table 2, representing "no" and "yes" Heart Problem are rough sets.) Although there is some ambiguity on what exactly is a "rough set" in the literature, this can be considered a standard definition of a rough set.

Generally, there are four different kinds of situations of rough sets based on whether $\underline{S}(X) = \varnothing$ and whether $\overline{S}(X) = U$, as can be defined as follows:

1. $\underline{S} \neq \varnothing$ and $\overline{S} \neq U$ then X is **roughly definable** (Fig. 6.7a).
2. $\underline{S} \neq \varnothing$ and $\overline{S} = U$ then X is **internally definable** (**or externally undefinable**) (Fig. 6.7b).
3. $\underline{S} = \varnothing$ and $\overline{S} \neq U$ then X is **externally definable** (or **internally undefinable**) (Fig. 6.7c).
4. $\underline{S} = \varnothing$ and $\overline{S} = U$ then X is **totally non-definable** (or **totally undefinable**) (Fig. 6.7d).

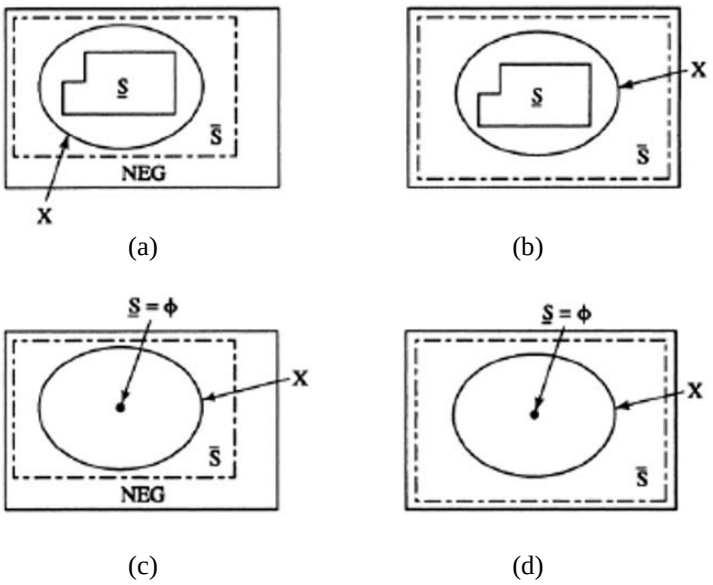


Fig. 6.7 Four types of definable and undefinable situations of rough sets. (a) Roughly definable. (b) Internally definable. (c) Externally definable. (d) Totally nondefinable.

The following is an intuitive meaning of the above classification in terms of the positive, boundary, and negative regions.

1. Roughly definable. There are elements in U that definitely belong to X (they are the elements in the positive region). Similarly, there are elements in U for which we can say that they definitely do not belong X, i.e., they definitely belong to -X, the complete of X (they are the elements in the negative region).
2. Internally definable. There are elements in U that definitely belong to X, but

there are no elements in U for which we can assure that they do not belong to X (since there is no negative region).

3. Externally definable. This is the opposite of internally definable. There are no elements in U for which we can say that they definitely belong to X (since there is no positive region), but there are elements in U for which we can say that they do not belong to X.
4. Totally non-definable. We cannot decide for any element of U whether it definitely belongs to X or -X.

Properties of $\underline{S}$ and $\overline{S}$

Every union of elementary sets (i.e., equivalence classes) is definable. This is because in this case, $\underline{S} = \overline{S} = X$ and BND = $\varnothing$, and by the definition of definable. Also, the following properties hold:

1. $\underline{S}(\varnothing) = \overline{S}(\varnothing)$
   $\underline{S}(U) = \overline{S}(U)$
2. $\underline{S}(X) \subseteq X \subseteq \overline{S}(X)$
3. $\underline{S}(X \cup Y) \supseteq \underline{S}(X) \cup \underline{S}(Y)$
4. $\underline{S}(X \cap Y) = \underline{S}(X) \cap \underline{S}(Y)$
5. $\overline{S}(X \cup Y) = \overline{S}(X) \cup \overline{S}(Y)$
6. $\overline{S}(X \cap Y) \subseteq \overline{S}(X) \cap \overline{S}(Y)$
7. $\underline{S}(U - X) = U - \overline{S}(X)$
8. $\overline{S}(U - X) = U - \underline{S}(X)$
9. $\underline{S}(\underline{S}(X)) = \overline{S}(\underline{S}(X)) = \underline{S}(X)$
10. $\overline{S}(\overline{S}(X)) = \underline{S}(\overline{S}(X)) = \overline{S}(X)$

6.5 Knowledge Representation Systems

A formal definition of knowledge representation systems (KRS)

In rough set theory a knowledge representation system (KRS) is formally defined as a quintuple as follows. The quintuple is an aggregate of objects, attributes and their

values, and a function. The KRS are typical applications of rough sets.

$$K = (U, C, D, V, \rho),$$

where:

U is a set of objects;
C is a set of condition attributes;
D is a set of action (or decision) attributes;
$V = \cup_{a \in F} V_a$, and V_a is the domain of attribute $a \in F$, where $F = C \cup D$;
ρ: $U \times F \rightarrow V$ for every $u \in U$ and $a \in F$ is an information function. ρ can also be represented as: ρ_u: $F \rightarrow V$ by $\rho_u(a) = \rho(u, a)$ for every $u \in U$ and $a \in F$.

The information function ρ can be defined in a more restricted way as: ρ: $U \times a \rightarrow V_a$ for every $u \in U$ and $a \in F$. The previous definition of ρ: $U \times F \rightarrow V$ can theoretically include many invalid mappings since V contains all possible values of all attributes. For example, (Adams, temperature) $\rightarrow$ "none" is probably invalid mapping; "none" is probably a value of another attribute, say, coughing rather than temperature. The second definition, ρ: $U \times a \rightarrow V_a$, eliminates such invalid mapping. Which definition to employ is one's choice, probably depending on a specific problem. If there are no errors, invalid mapping will not occur. If $|V|$ is small, the use of V is manageable.

Generally, how to represent human knowledge is a difficult problem and it is a major problem in AI research today. Human knowledge is so complex, there probably is no simple answer on what is the best form to represent it. Two most popular methods for representing knowledge in knowledge-based systems are rule-based and frame-based. The KRS here is closely related to the rule-based systems. The condition, premise, or "if" part of an if-then rule corresponds to the condition attributes, and the action, conclusion, or "then" part of the if-then rule corresponds to decision attributes in this KRS.

As an example, consider a problem of representing "knowledge" on how various factors affect heart and stroke problems of persons. Many factors are conceivable, some have obvious close connections to these diseases while some don't. Some possible factors are: laboratory test results, such as temperature, pulse, blood pressure, blood test (e.g., good and bad cholesterol), urinalysis, EKG, etc.; symptoms such as chest pain, dizziness, etc.; diet, i.e., what to eat and drink; and so forth. Then C, set of condition attributes, is the set of these factors, for example, C = {temperature, pulse, . . ., EKG}. (For simplicity, we have dropped symptoms and diet, etc.) If there are 25 of these condition attributes, $|C| = 25$. The set of decision attributes, D has two elements, D = {heart_problem, stroke_problem}, and $|D| = 2$. $F = C \cup D$ in this case is F = {temperature, pulse, . . ., EKG; heart_problem, stroke_problem} and $|F| = 27$. Let us assume that the first attribute temperature is measured as either normal, slightly_high, or high. Then V_a for attribute a = temperature is the domain of the values of this attribute, i.e., $V_{\text{temperature}}$ = {normal, slightly_high, high}. Although $|V_{\text{temperature}}| = 3$ in this example, $|V_a|$ can be any value. Similarly, we may define other domains, for example, V_{pulse} = {low, medium, high}, . . , $V_{\text{stroke_problem}}$ = {none, moderate, high}. V is the union of all these V_a's, e.g., V = {normal, slightly_high, high, low, medium, . . .}. A short note on the

definition on V: The elements of V are distinct since V is a set. Hence, for example, "high" in V_{pulse} = {low, medium, high} will not appear in V repeatedly after "high" for $V_{temperature}$ as V = {normal, slightly_high, high, low, medium, high, . . .}. This means that $|V| \leq \Sigma_{a \in F} |V_a|$. In certain applications, it may be desirable to distinguish high pulse from high temperature. In such case, we can define $V_{temperature}$ = {normal_temperature, slightly_high_temperature, high_temperature} and V_{pulse} = {low_pulse, medium_pulse, high_pulse}.

Suppose that information for these condition and decision attributes is collected for 10,000 men. Some of the information can be unknown or inaccurate. In such a case, for example, "unknown" can be added as a possible attribute value. For example, there may not be any EKG test result for Man No. 7,825. The 10,000 men are the elements of set U, which can be represented as $U = \{u_1, u_2, \ldots, u_{10,000}\}$. Now we can construct the cartesian product $U \times F$. In set form, $U \times F$ is the set of many ordered pairs as $\{(u_1$, temperature), . . ., ($u_{10,000}$, stroke_problem)}. (How many elements are in $U \times F$? Yes, there are 270,000.) Or, in matrix or two-dimensional array form, we have 10,000 rows corresponding to the 10,000 men and 27 columns corresponding to the 25 conditional plus 2 decision attributes.

Now by using the collected information of 270,000 values (of which some may have unknown values) for the 10,000 men, we can fill in the $U \times F$ matrix, as e.g., (u_1, temperature) = slightly_high, . . ., ($u_{10,000}$, stroke_problem) = none. Each row of this matrix representing a specific man is called an **object** (or **entity**). This matrix, with the specific element values, can be viewed as mapping from $U \times F$ to V, and denoted as an information function, ρ: $U \times F \rightarrow V$. Combining all these values, we have our KRS as $K = (U, C, D, V, \rho)$. (During the time this part of the book was being prepared, a Harvard report in the *New England Journal of Medicine* was just published. It said that eating fish does not particularly help to avoid heart problems based on the study for 45,000 men for six years. Can you think of how this study can be formulated as a KRS?)

We note that our KRS format can represent many types of applications. Here are some examples:

Diagnosis:	The above heart and stroke problem is in this category. The same concept can be applied to engineering, business, and other medical problems.
Prediction:	e.g., in finance to predict a stock market.
Control:	Simply stated, control is input-output mapping. For example, to control room temperature, the temperature from a sensor is input and a heat or cooling source is output. The conditional attributes correspond to input and the decision attributes to output.
Machine learning:	Discovering from information functions.

A knowledge representation system (KRS) example

We will consider the following fictitious example to illustrate a knowledge representation system.

U is a set of 8 persons (objects): $U = \{u_1, u_2, ..., u_8\}$. Here u_1, u_2,..., are symbolic representations of Adams, Brown, and so on.
C is a set of 3 condition attributes: C = {Temp, Blood-P (Blood-Pressure), Vision (Eyesight)}
D is a set of 2 decision attributes: D = {Heart-Risk, Health (General-Health)}

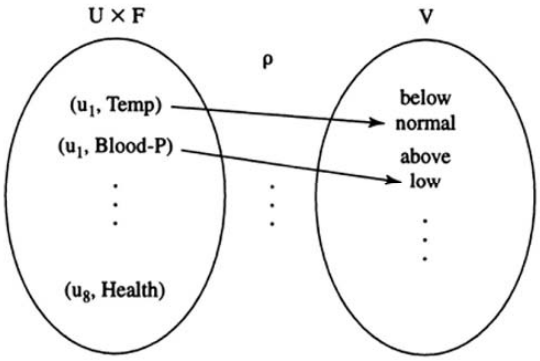


Fig. 6.8. An information function ρ from $U \times F$ to V.

The domains of individual attributes are given by:

V_{Temp} = {below, normal, above}
$V_{\text{Blood-P}}$ = (low, average, high}
V_{Vision} = {near, standard, far}
$V_{\text{Heart-Risk}}$ = {none, slight, serious}
V_{Health} = {poor, good, excellent}

We have:

$F = C \cup D$ = {Temp, Blood-P, Vision; Heart-Risk, Health}
$V = \cup_{a \in F} V_a$ ={below, normal, above; low, average, high; near, standard, far; none, slight, serious; poor, good, excellent}
ρ: $U \times F \rightarrow V$, i.e., ρ: {(u_1, Temp),..., (u_8, Health)} $\rightarrow$ {below,..., excellent} is an information function (Fig. 6.8).

The following Table 3 is one of such information functions.

Table 3. An example information function in information table form for a knowledge representation system

U	C			D	
Person	Temp	Blood-P	Vision	Heart-Risk	Health
u_1	normal	low	far	slight	poor
u_2	below	average	standard	serious	excellent
u_3	above	low	near	serious	good
u_4	normal	average	near	slight	excellent
u_5	normal	low	far	none	good
u_6	above	high	near	serious	good
u_7	above	average	standard	serious	excellent
u_8	below	average	standard	none	good

6.6 More on the Basics of Rough Sets

Equivalence relations on attributes

As discussed before, for any subset G of C or D, we can define an equivalence relation $\tilde{G}$ on U such that $(u_i, u_j) \in \tilde{G}$ if and only if $\rho(u_i, g) = \rho(u_j, g)$ for every $g \in G$. That is, $\tilde{G} = \{(u_i, u_j) \mid u_i$ and u_j have the same value for every attribute $g \in G\}$. In words, we group the elements based on the values of specific attributes; elements u_i and u_j are related if all the values of the specific attributes are the same, not related otherwise. We denote the partition induced by the equivalence relation $\tilde{G}$ as G^*.

Example 7. (From Table 3)

$$G = \{\text{Temp}\}.$$

In this case, there is only one element (attribute) in G, which is denoted as g = Temp.

$\rho(u_1, \text{Temp}) = \rho(u_4, \text{Temp}) = \rho(u_5, \text{Temp}) = \text{normal}$
$\rho(u_2, \text{Temp}) = \rho(u_8, \text{Temp}) = \text{below}$
$\rho(u_3, \text{Temp}) = \rho(u_6, \text{Temp}) = \rho(u_7, \text{Temp}) = \text{above}$

Hence, $\tilde{G} = \{(u_1, u_1), (u_2, u_2), ..., (u_8, u_8); (u_1, u_4), (u_1, u_5), (u_4, u_5), (u_4, u_1), (u_5, u_1), (u_5, u_4), (u_2, u_8), (u_8, u_2), (u_3, u_6), (u_3, u_7), (u_6, u_7), (u_6, u_3), (u_7, u_3), (u_7, u_6)\}$.

Fig. 6.9 shows G^*, the partition of the three subsets of U induced by $\tilde{G}$.

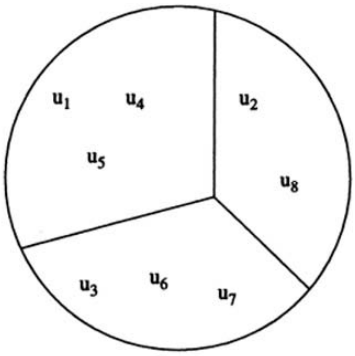


Fig. 6.9.The partition G^* induced by an equivalence relation $\tilde{G}$, where $(u_i, u_j) \in \tilde{G}$ if and only if $\rho(u_i, \text{Temp}) = \rho(u_j, \text{Temp})$ in Table 3.

Example 8. (From Table 3)

$$G = \{\text{Temp, Blood-P}\}$$

In addition to $\rho(u_i, \text{Temp}) = \rho(u_j, \text{Temp})$ in the previous example we have:

$\rho(u_1, \text{Blood-P}) = \rho(u_3, \text{Blood-P}) = \rho(u_5, \text{Blood-P}) = \text{low}$
$\rho(u_2, \text{Blood-P}) = \rho(u_4, \text{Blood-P}) = \rho(u_7, \text{Blood-P}) = \rho(u_8, \text{Blood-P}) = \text{average}$
$\rho(u_6, \text{Blood-P}) = \text{high}$

For (u_i, u_j) to be in $\tilde{G}$, $\rho(u_i, g) = \rho(u_j, g)$ must hold for every g in G, i.e., in our example, for both g = Temp and g = Blood-P. This means that we take the intersection of the two equivalence relations, one for Temp and the other for Blood-P:

$$\tilde{G} = \{ (u_1, u_1), (u_2, u_2), ..., (u_8, u_8); (u_1, u_5), (u_5, u_1), (u_2, u_8), (u_8, u_2)\}.$$

The partition induced by $\tilde{G}$ is the product of the partitions induced by the two equivalence relations for Temp and Blood-P: $G^* = \{X_1, X_2, X_3, X_4, X_5, X_6\}$, where $X_1 = \{u_1, u_5\}$, $X_2 = \{u_4\}$, $X_3 = \{u_2, u_8\}$, $X_4 = \{u_3\}$, $X_5 = \{u_6\}$, $X_6 = \{u_7\}$. Fig.6.10 shows the partition; the superposed dashed lines correspond to the partition induced by Blood-P.

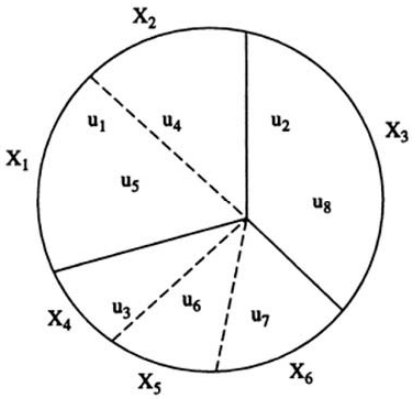


Fig. 6.10.The partition induced by an equivalence relation $\tilde{G}$, where $(u_i, u_j) \in \tilde{G}$ if and only if $\rho(u_i,$ Temp) = $\rho(u_j,$ Temp) and $\rho(u_i,$ Blood-P) = $\rho(u_j,$ Blood-P) in Table 3.

POS, BND, and NEG regions of a partition

Let $A \subseteq C$, i.e., A is a set of some condition attributes, as, e.g., A = {Temp, Blood-P}. Let $B \subseteq D$, i.e., B is a set of some decision attributes as, e.g., B = {Heart-Risk} or B = {Heart-Risk, Health}. Let $A^* = \{X_1, ..., X_n\}$ and $B^* = \{Y_1, ..., Y_m\}$ denotes the partitions on U induced by the equivalence relations $\tilde{A}$ and $\tilde{B}$, respectively. (These are for A and B in place of G in the previous example.) We will be interested in determining to what extent the partition B^* as a whole can be characterized or approximated by the partition A^* (Fig. 6.11).

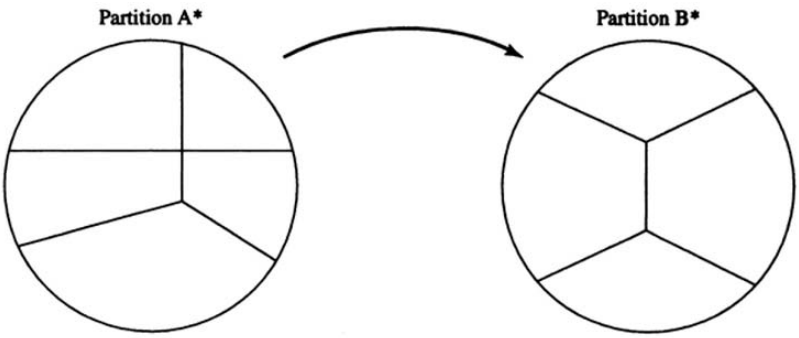
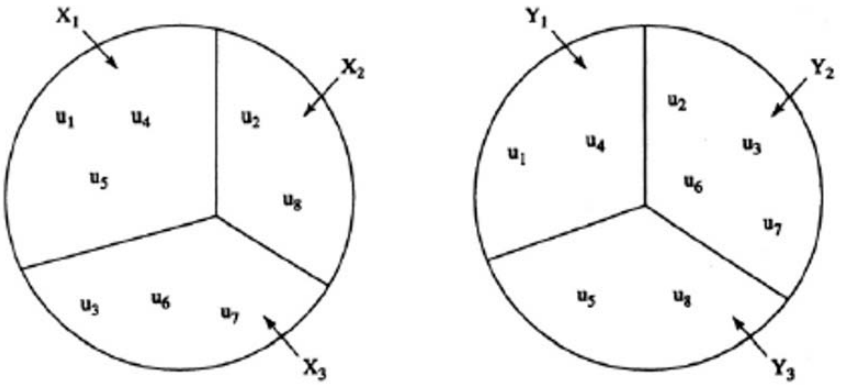


Fig. 6.11. The characterization of partition B^* on partition A^*.

Example 9. The partitions induced by A = {Temp} and B = {Heart-Risk} in Table 3.

Let A = {Temp}. As we saw before, $\tilde{A} = \{(u_1, u_1), (u_2, u_2), ..., (u_8, u_8); (u_1, u_4), (u_1, u_5),$

(u_4, u_5), (u_4, u_1), (u_5, u_1), (u_5, u_4), (u_2, u_8), (u_8, u_2), (u_3, u_6), (u_3, u_7), (u_6, u_7), (u_6, u_3), (u_7, u_3), $(u_7, u_6)\}$ and $A^* = \{X_1, X_2, X_3\}$ is given in Fig. 6.12(a). Let B = {Heart-Risk}. Then similarly, $\tilde{B}$ = $\{(u_1, u_1), (u_2, u_2), ...\}$, and $B^* = \{Y_1, Y_2, Y_3\}$ is given in Fig. 6.12(b).



(a) (b)

Fig. 6.12 (a) The partition $A^* = \{X_1, X_2, X_3\}$ induced by $\tilde{A}$ where A = {Temp}. (b) The partition $B^* = \{Y_1, Y_2, Y_3\}$ induced by $\tilde{B}$ where B = {Heart-Risk}.

In terms of the lower approximation $\underline{S}(Y_j)$ and upper approximation $\overline{S}(Y_j)$ of $Y_j \in B^*$ in the approximation space $S = (U, \tilde{A})$, we define the **positive**, **boundary**, and **negative regions** of the **partition** B^*, as follows:

$$\text{POS}_S(B^*) = \cup_{Yj \in B^*} \underline{S}(Y_j) = \cup_{Yj \in B^*} [\cup_{Xi \subseteq Yj} X_i]$$

$$\text{BND}_S(B^*) = \cup_{Yj \in B^*} (\overline{S}(Y_j) - \underline{S}(Y_j)) = \cup_{Yj \in B^*} [\cup_{Xi \cap Yj \neq \varnothing} X_i - \cup_{Xi \subseteq Yj} X_i]$$

$$\text{NEG}_S(B^*) = U - \cup_{Yj \in B^*} (\overline{S}(Y_j)) = U - \cup_{Yj \in B^*} [\cup_{Xi \cap Yj \neq \varnothing} X_i]$$

Note that $\text{POS}_S(B^*)$, $\text{BND}_S(B^*)$, and $\text{NEG}_S(B^*)$ defined above are distinct.

Note also that the argument type of B^* in $\text{POS}_S(B^*)$, etc. defined here is different from the argument type of X in $\text{POS}_S(X)$, etc. defined in Section 6.4. X is a set of elements while B^* is a partition induced by an equivalence relation, i.e., a set of sets of elements.

We may understand these as our way of defining POS, etc. That is, for X, we define $\text{POS}_S(X) = \underline{S}(X)$. For B^*, we define $\text{POS}_S(B^*)$ = union of $\underline{S}(X_i)$ where $X_i \in B^*$. The latter case can be described more generally as: if $F = \{X_1, X_2, ..., X_n\}$, then $\text{POS}_S(F) = \text{POS}_S(X_1) \cup \text{POS}_S(X_2) \ldots \cup \text{POS}_S(X_n)$. In both cases of $\text{POS}_S(X)$ and $\text{POS}_S(B^*)$, the results are sets of elements, i.e., they are of the same type and consistent, and will not cause any problem. Incidentally, this type of situation is common in mathematics. For example, a function may be defined on different types

of arguments, one for a scalar, the other for a vector, or a matrix. Of course, the function must be defined for each of the different types of arguments.

Example 10. $POS_S(B^*)$, $BND_S(B^*)$, and $NEG_S(B^*)$ applied to Example 9.

$\underline{S}(Y_1) = \cup_{Xi \subseteq Y1} X_i = \varnothing$, $\underline{S}(Y_2) = X_3$, $\underline{S}(Y_3) = \varnothing$

$\overline{S}(Y_1) = \cup_{Xi \cap Y1 \neq \varnothing} X_i = X_1$, $\overline{S}(Y_2) = X_2 \cup X_3$, $\overline{S}(Y_3) = X_1 \cup X_2$

$POS_S(B^*) = \cup_{Yj \in B^*} \underline{S}(Y_j) = \underline{S}(Y_1) \cup \underline{S}(Y_2) \cup \underline{S}(Y_3) = \varnothing \cup X_3 \cup \varnothing = X_3$

$BND_S(B^*) = \cup_{Yj \in B^*} (\overline{S}(Y_j) - \underline{S}(Y_j)) = (X_1 - \varnothing) \cup (X_2 \cup X_3 - X_3) \cup (X_1 \cup X_2 - \varnothing) = X_1 \cup X_2$

$NEG_S(B^*) = U - \cup_{Yj \in B^*} (\overline{S}(Y_j)) = U - ((X_1) \cup (X_2 \cup X_3) \cup (X_1 \cup X_2)) = \varnothing$ □

Attribute dependencies

Various measures can be defined to represent how much *B*, a set of decision attributes, depends on *A*, a set of condition attributes. In the following, we state some of these measures. Probably the most common measure is the dependency.

Dependency $\gamma_A(B)$

The **dependency** of *B* on *A*, denoted as $\gamma_A(B)$, is a plausible measure of how much *B* depends on *A* and is defined as follows.

$$\gamma_A(B) = | POS_S(B^*) | / | U |$$

where $S = (U, \tilde{A})$ is the approximation space, and $| \ |$ denotes the cardinality (i.e., the number of elements) of a set. Note that $0 \leq \gamma_A(B) \leq 1$. In particular,

1. $\gamma_A(B) = 1$: *B* is **totally dependent** on *A*, i.e., A functionally determines *B*.
2. $\gamma_A(B) = 0$: *A* and *B* are **totally independent** of each other.
3. $0 < \gamma_A(B) < 1$: *B* is **roughly dependent** on *A*.

In general, the dependency of *B* on *A* can be denoted by $A \rightarrow_\gamma B$. For example, $A \rightarrow_1 B$ if *B* is totally dependent on *A*.

Example 11. $\gamma_A(B)$ applied to Example 10.

$$\gamma_A(B) = | POS_S(B^*) | / | U | = | X_3 | / | U | = 3/8 = 0.375.$$

i.e., {Temp} $\rightarrow_{0.375}$ {Heart-Risk}.

Example 12. (From Table 3)

$$\{\text{Temp, Blood-P, Vision}\} \rightarrow_{0.5} \{\text{Heart-Risk}\}$$
$$\{\text{Temp, Blood-P}\} \rightarrow_{0.5} \{\text{Heart-Risk}\}$$

We can say that to determine {Heart-Risk}, the {Temp, Blood-P} or {Temp, Blood-P, and Vision} knowledge is not sufficient, since $\gamma = 0.5$. Also, {Vision} is superfluous since the removal does not affect the dependency.

As a special case of the dependency, when we choose A as a set of one condition attribute a as $\gamma_{\{a\}}(B)$, it is a measure of how much B depends on that specific condition attribute. That is, $\gamma_{\{a\}}(B)$ gives the importance level of a in determination of B. In the following, we simply state the definitions of other measures.

Discriminant index $\beta_A(B)$

$$\beta_A(B) = \left| \text{POS}_S(B^*) \cup \text{NEG}_S(B^*) \right| / \left| U \right| = \left| U - \text{BND}_S(B^*) \right| / \left| U \right|$$

The **discriminant index** is a measure of the degree of certainty in determining whether elements in U are elements of B or not B. This is can also be interpreted as a measure indicating how much uncertainty can be removed by selecting $S = (U, \tilde{A})$.

Significance $\sigma_A(B)$

The **significance** of B on a specific condition attribute a can be defined by using the dependencies as follows:

$$\sigma_{\{a\}}(B) = \gamma_C(B) - \gamma_{C-\{a\}}(B)$$

In words, the significance of B on a is the difference between the dependency of B on the set of all condition attributes C and the dependency of B on the set of all condition attributes *except* the specific attribute a. That is, the significance measures the importance level of the attribute by considering how a deletion of the attribute from the entire set of condition attributes affects the dependency. $\gamma_{C-\{a\}}(B)$ is a sort of the "complement dependency" of $\{a\}$ with respect to C.

We can further extend the significance for $\{a\}$ to A, a set of any number of condition attributes:

$$\sigma_A(B) = \gamma_C(B) - \gamma_{C-A}(B)$$

This is a measure to indicate the importance of the set of condition attributes A, rather than a single attribute a.

The dependency and discriminant index are "direct" measures focusing on one or more condition attributes. On the other hand, the significance is "complementary," that is, it considers the entire set of condition attributes. Which measure or measures are to be used depends on a specific application. For example, if $\gamma_{\{a\}}(B)$ is equal to 1, then B is totally depends on a. Hence, no other measures may need to be determined.

Reducts and elimination of superfluous attributes

In a knowledge representation system, each entity is described by the attribute values of C, the set of condition attributes. (For example, in Table 3, Object u_1, is described by: Temp = normal, Blood-P = low, and Vision = far.) Some attributes in C can be redundant and thus can be eliminated.

Let B be a non-empty subset of C. B is called a **dependent set** of attributes if there exists a proper subset $B' \subset B$ such that $\tilde{B}' = \tilde{B}$, i.e., $B' \rightarrow_1 B$; otherwise, B is called an **independent set** or **minimal set**. B is said to be a **reduct** of C if B is a maximal independent set of condition attributes. (**Maximal** means an addition of any attribute to B would make the new B dependent.) A reduct of C is denoted as $\hat{C}$. $\hat{C}$ induces the same partition as C. In general, more than one reduct of C can be identified. The **collection of all reducts of *C*** is denoted by RED(C).

For example, consider the following Table 4.

Table 4. Illustration of dependent/independent sets, reducts and the core

U	C					D
Person	Temp	Blood-P	Vision	EKG	Cholesterol	Heart-Risk
u_1	normal	low	far	. . .	. . .	slight
u_2	below	average	standard	. . .	. . .	serious
.	. . .	. . .	. . .	. . .	. . .	. . .
.	. . .	. . .	. . .	. . .	. . .	. . .

Here we consider C = {Temp, Blood-P, Vision, EKG, Cholesterol} (but not D = {Heart-Risk}). Let B_1 = {Temp, Blood-P, Vision} and B_2 = {Temp, Blood-P}. Suppose that B_1 and B_2 leads to the same equivalence relation on U, i.e., they induce the same partition on U; then B_1 is a dependent set. B_2 is an independent set if there is no proper subset of B_2 that has the same equivalence relation. In other words, B_2 is an independent set if a deletion of any attribute of B_2 results in a different equivalence relation. Furthermore, B_2 is a reduct of C if B_2 is a maximal independent set, i.e., an addition of any attribute such as Vision, EKG, or Cholesterol, would make the set dependent. In other words, B_2 has the same equivalence relation as C, i.e., it induces the same partition as C. Suppose that B_3 = {Blood-P, EKG, Cholesterol} is another reduct of C, and B_2 and B_3 are the only reducts of C. Then RED (C) = {B_2, B_3}.

We can extend the above definitions, dependent and independent sets, and reduct, to take into account the set of decision attributes D. We do this with the notion of positive regions. Let again B be a non-empty subset of C. B is called a **dependent set with respect to** D if there exists a proper subset $B' \subset B$ such that $\text{POS}_{B'}(D^*) = \text{POS}_B(D^*)$; otherwise, B is regarded as an **independent set with respect to** D. B is said to be a **relative reduct** of C if B is a maximal independent set with respect to D. The collection of all such relative reducts is denoted by $\text{RED}_D(C)$. We note that for any reduct or relative reduct $\hat{C}$ of C, $C \rightarrow_\gamma D$ always implies $\hat{C} \rightarrow_\gamma D$, i.e., C can be

reduced to $\hat{C}$, without a loss of information.

For example, in Table 4, these extensions will take into account set D = {Heart-Risk} in terms of $POS_B(D^*)$. For example, B_2 = {Temp, Blood-P} is an independent set with respect to D, if a deletion of any attribute results in different $POS_B(D^*)$. B_2 is a relative reduct of C if $POS_{B2}(D^*) = POS_C(D^*)$. Suppose that B_2 and B_3 are the only relative reducts of C. Then $RED_D(C) = \{B_2, B_3\}$.

Example 13. (From Table 3)

$\hat{C}$ = {Temp, Blood-P} is the only relative reduct of C = {Temp, Blood-P, Vision}. C can be reduced to $\hat{C}$; Table 3 can be transformed to another equivalent and simpler table (Table 5).

Table 5. Reduced knowledge representation system

U	C		D	
Person	Temp	Blood-P	Heart-Risk	Health
u_1	normal	low	slight	poor
u_2	below	average	serious	excellent
u_3	above	low	serious	good
u_4	normal	average	slight	excellent
u_5	normal	low	none	good
u_6	above	high	serious	good
u_7	above	average	serious	excellent
u_8	below	average	none	good

The **core** of C is defined as the set of condition attributes belonging to the intersection of all reducts of C:

$$\text{CORE}(C) = \cap_{B \in \text{RED}(C)} B$$

For example, suppose C = {Temp, Blood-P, Vision, EKG, cholesterol}, and only reducts of C are {Temp, Blood-P, EKG} and {Blood-P, EKG, cholesterol}. Then CORE (C) = {Blood-P, EKG}.

When a deletion of a condition attribute from C results in a different equivalence relation from the equivalence relation defined for C, then the condition attribute is called **indispensable**. In equation form, a condition attribute $a \in C$ is **indispensable** if $\tilde{C}_a \neq \tilde{C}$ where $C_a = C - \{a\}$. The core of C is equal to the set of all indispensable attributes in C. When dealing with an information table, a common problem is to identify the most essential condition attributes. The core or all of its elements, the indispensable condition attributes, is *necessary* in order to have the same equivalence relation as C, although it is *not sufficient*. A reduct is sufficient to have the same equivalence relation as C. When there are many reducts, a selection of a reduct is not necessarily obvious. We can employ various criteria such as selecting a reduct of the

smallest number of attributes, or selecting a reduct that contains most common attributes for the specific application under consideration, and so forth.

As before, we can extend the above definitions to take into account D, the set of action attributes. The **relative core** is the set of condition attributes belonging to the intersection of all relative reducts of C:

$$\mathrm{CORE}_D(C) = \cap_{B \in \mathrm{RED}D(C)} B$$

A condition attribute $a \in C$ is said to be **indispensable with respect** to D if $\mathrm{POS}_{c-\{a\}}(D^*) \neq \mathrm{POS}_c(D^*)$. The relative core of C is equal to the set of all indispensable condition attributes with respect to D. The core can be easily determined from a KRS. The core is a subset of every reduct, i.e., every reduct is a superset of the core. Hence, it is advantageous to start with the core in order to find a reduct.

Example 14. (From Table 3)

$$C = \{\text{Temp, Blood-P, Vision}\},\ D = \{\text{Heart-Risk, Health}\}.$$

Let $D' = \{\text{Heart-Risk}\}$. Then the relative core of C with respect to D':

$$\mathrm{CORE}_{D'}(C) = \{\text{Temp}\}$$

There are two relative reducts of set C with respect to D':

$$B_1 = \{\text{Temp, Blood-P}\},\ B_2 = \{\text{Temp, Vision}\},$$

i.e., $\mathrm{RED}_{D'}(C) = \{B_1, B_2\}$, and $\mathrm{CORE}_{D'}(C) = B_1 \cap B_2$.

6.7 Additional Remarks

Implementation considerations

There are many ways to employ rough sets depending on specific applications. The following are some possible considerations.

1. Representation of input data and information tables. Since rough sets deal with information tables, computer processing requires storing raw data represented as information tables. The simplest data structure for an information table will be a two-dimensional array since the table is a two-dimensional matrix. When the size of the table is not known in advance, however, a linked list is more flexible to dynamically allocate memory space. A linked list with pointers of row-wise and column-wise can be used to access horizontal and vertical directions of the table. Typically, an array may be easier for programming, while a linked list may be more flexible for variable size of information tables.

2. Preparation and analysis of input data. Input data can be carefully prepared

manually or sometimes through other pre-processing techniques such as statistical analysis. We try to include the minimum necessary and sufficient information for a particular application. The condition attributes can be manually arranged in decreasing order of importance, if it is known.

3. Discretization of input data. Raw data is often given as numeric values, such as 99.7, rather than descriptive ones given in examples in this chapter, as, for example, Temp = below, normal, or above. When raw data is in numeric form, usually we need to pre-process it by assigning one of discrete intervals - a sort of "quantization" of continuous data. Such quantization process is called **discretization**. For a programming purpose, descriptive values such as "below" do not necessarily have to be used exactly as they appear. Instead, discrete numeric values can be associated as, for example, below = 1, normal = 2, and above = 3. This numeric representation is often easier for programming.

4. Analysis of condition attributes. For example, determining indispensable attr ibutes, reducts, and the core (with or without respect to a decision attribute).

   Suppose that we arrange the condition attributes in decreasing order of importance, based on intuition or some sort of pre-processing. We can start from the least important attribute. We drop it and check whether the resulting partition is the same as the original one containing all the condition attributes. This requires exhaustive comparisons by the computer. If the result is the same, the attribute is dispensable; otherwise it is indispensable. We can repeat this process for the remaining attributes, each time by choosing one attribute. At the end of this process, we know whether each attribute is dispensable or indispensable for the entire set of condition attributes. The set of all the indispensable attributes is the core, that is, the set of absolutely necessary condition attributes (although it may not be sufficient).

   A reduct is a set of sufficient condition attributes equivalent to the original data and we can work on it to derive rules. Several scenarios are possible to determine a reduct depending on a specific application.

   (a) If all the attributes are indispensable, we cannot drop any of the attributes. The core is the only reduct.
   (b) If some attributes are indispensable while others are not, we can start from the core to find a reduct. We pick one of the dispensable attributes, and keep adding one at a time, until the resulting partition is the same as the original one with all the input attributes. This process requires exhaustive comparisons.
   (c) In another extreme case, there may be no indispensable attributes, that is, the core is empty. In this case, we can start from a set of some dispensable attributes, adding or dropping one at a time until we find a reduct.

   In practice, often it is necessary to limit the number of attributes and the number of possible values each attribute can take. For example, if there are seven attributes and each attribute can take one of four values, the number of possible combinations is 4 ** 7 = 16,384. This number may be too big; we may

want to limit the maximum to, say, 10,000.

For certain applications, the implementation can stop here. The resulting table of this Step 4 can be directly used to obtain, for example, condition to decision attribute mapping.

5. Determination of relations (e.g., POS) of the decision attributes on the condition attributes. This step may simplify the condition-to-decision mapping. More are given in a separate topic below.

   Again, for certain applications, the implementation can stop here. The result of this Step 5 can be directly used for certain types of applications.

6. Induction of rules. We can write explicit rules based on the information obtained in Step 5 (e.g., if the input variable 1 = small, . . ., then the decision variable = medium). Simplification of the set of the rules (e.g., combining rules) can be done on the table form in Step 5 before deriving rules. This approach may typically be easier than deriving simple rules first, then trying to simplify the set of the rules.

Determination of relations

There are two major approaches to determine POS regions, etc., of the decision attributes on the condition attributes.

1. Local-to-global approach : In this approach, we start with single condition attributes (which are considered to be "local") to determine the relations and gradually increase the number of attributes. That is, we work on lower-to-higher number of condition attributes. For example, suppose that there are three condition attributes, a_1, a_2, and a_3. In Pass 1, we consider one condition attribute at a time, and determine the relations. For example, if a_1 = low defines a positive region for decision = high, this can be a rule. For a_1 = low, we need not add other attributes as, for example, a_1 = low, a_2 = low, and a_3 = ... For those cases where POS are not obtained, we can proceed with a higher number of condition attributes. In Pass 2, we will consider two condition attributes at a time. For example, a_1 = medium and a_3 = high may be POS for decision = low, which can be a rule. Again, we need not consider a_2, since the rule should hold for all possible values of a_2. In Pass 3, we consider a_1, a_2, and a_3 for the remaining cases, which are left out from preceding passes.

   The rules derived by the above process can be combined, yielding a smaller number of rules. How to combine the rules may depend on a specific application. When we gradually increase the number of condition attributes, we must make certain that there are no redundant attributes. That is, we must make certain that the set of condition attributes is not larger than a reduct.

2. Global-to-local approach: This is the opposite of approach 1, that is, we work on higher-to-lower number of condition attributes. Suppose that we have a reduct. We determine the relations on partitions derived from considering all the condition attributes in the reduct (which is considered as "global" as a whole). The partitions will be finest in structure in this process. After determining

positive regions for these partitions, we may find certain condition attributes can be dropped for certain rules because the values of these attributes do not contribute to the rules. For example, suppose that we have "if a_1 = low and a_2 =low (and a_3 = low or medium or high) then decision = low". Then a_3 can be dropped since the rule does not depend on a specific value of a_3, leading to a generalized rule: "if a_1 = low and a_2 then decision = low".

3. As an extension of the above, these two approaches can be mixed. For example, start with single condition attributes to a certain number of attributes, then switch to the global-to-local approach for the remaining cases.

Extensions of the deterministic rough sets

The model discussed in the previous sections is based on **deterministic** information. This model can be extended to include **probabilistic** information to deal with the non-deterministic problems. Or, the model can be extended to incorporate **fuzzy logic** - a hybrid system of rough and fuzzy sets. These extended models are particularly suitable for **uncertainty** problems.

The basic idea of these extensions is to associate the probability or fuzziness to the information dealt with. For example, health of a person may not be simply poor or good. Instead, it may be poor and good with a certain probability or fuzziness.

Major uses of rough sets

Major types of functions rough sets can perform include: data reduction (i.e., elimination of superfluous information); discovering of condition-action dependencies, and approximate classification of data. These types of operations lead to the approximation domains such as knowledge-based systems and application areas such as engineering, discussed before at the beginning of this chapter.

6.8 Case Study and Comparisons with Other Techniques

Comparing rough set theory and other techniques in a general term is a difficult task. There have been debates on this issue, and they will continue in the future. While there are no mathematical proofs to show which technique is most suitable for what types of problems, it appears that each technique has specific strengths and limitations for certain problems. Similar situations also exist for other types of problems. In case of optimization, for example, many techniques are employed depending the types of the problems including analytical methods, operations research techniques (such as linear programming, dynamic programming and various heuristic approaches), so-called guided random search techniques (such as neural networks and genetic algorithms), and so on.

In this section, we will put aside this difficult issue and start with a simple case study to apply rough set theory. We then apply a decision tree technique called ID3 to the same case study. ID3 is the most successful machine learning technique in traditional AI in terms of practical applications. Finally, we will briefly compare

rough set theory with some other techniques.

6.8.1 Rough Sets Applied to the Case Study

Case study - process control

Control is often the most successful domain for practical applications in many areas in AI. Control here refers to the control of physical characteristics such as temperature, pressure, speed, and electric current, or chemical characteristics such as ingredients of raw materials, and so forth. There are different types of AI applications to the control problem. One is developing efficient and robust control for difficult problems. Fuzzy control falls into this category. Another type is automatic induction of control rules from sampled input-to-output mappings. Neural networks have been applied to these types of control problems, especially for numeric data. In this case study, we consider automatic induction of process control rules for material production at a plant. For example, production of fuel for nuclear power plants is commonly performed by transforming uranium hexafloride gas into pellets of uranium dioxide powder. These pellets must be of high quality, but complex interactions among the parameters (attributes) make prediction of the quality difficult. Conventional statistical methods may have limited success and machine learning techniques may give significant contributions.

In the following, we set up a fictitious, much simplified version of process control. The sampled raw data is given as Table 6. The number of samples in a real world problem would be much higher, say, 5,000, rather than 10. Our problem is to induce rules of condition-to-decision, or input-to-output in control terms, based on this information.

Table 6. Sampled raw data for process control.

U	C		D	
Sample	Temp	Pressure	Size	Quality
u_1	normal	low	fine	good
u_2	normal	low	fine	good
u_3	above	medium	coarse	bad
u_4	normal	medium	coarse	bad
u_5	normal	medium	fine	good
u_6	above	high	coarse	bad
u_7	above	high	fine	bad
u_8	normal	high	fine	bad
u_9	above	high	coarse	good
u_{10}	normal	high	fine	bad

For example, u_1 represents one of 10 samples or batches actually taken at the plant. When the processing temperature was normal, the pressure was low, and the grain

size of the raw material was fine, the quality of the product was good.

Rough sets approach

The partition induced by all the condition attributes, Temp, Pressure and Size is $\{\{u_1 u_2\}, \{u_3\}, \{u_4\}, \{u_5\}, \{u_6, u_9\}, \{u_7\}, \{u_8, u_{10}\}\}$. All the condition attributes are indispensable, that is, a deletion of any attribute will result in a different partition. Hence, the set of all the condition attributes is the core as well as the only reduct. We will work on this reduct by the local-to-global approach. In this case, we need not check redundancy in the process of adding condition attributes, since they are all indispensable. If we know the importance ranking of the condition attributes, we can rearrange them in that order. We assume that we do not have this information in advance, therefore we keep the original order of Temp, Pressure, and Size.

Cases of one condition attribute

We select one condition attribute at a time, partition the universe based on the attribute, then determine the positive regions with respect to $D = \{Quality\}$.

	Quality	
Temp	good	bad
normal	$u_1\ u_2\ u_5$	$u_4\ u_8\ u_{10}$
above	u_9	$u_3\ u_6\ u_7$

We see no positive regions.

	Quality	
Pres	good	bad
low	$\underline{u_1\ u_2}$	
medium	u_5	$u_3\ u_4$
high	u_9	$u_6\ u_7\ u_8\ u_{10}$

$\{u_1, u_2\}$ is a positive region (underlined); we can have a certain rule, as for example:

Rule 1: if Pressure = low, then Quality = good

	Quality	
Size	good	bad
fine	$u_1\ u_2\ u_5$	$u_7\ u_8\ u_{10}$
coarse	u_9	$u_3\ u_4\ u_6$

There are no positive regions. At the end of cases of one condition attribute, we can rearrange the order of the attributes. In this example, we can place Pressure first since it appears to play a key role for decision making. We will place Pressure, Temp, and Size, in this order.

Cases of two condition attributes

There are three combinations of the attributes, (Pressure, Temp), (Pressure, Size), and (Temp, Size).

		Quality	
Pres	Temp	good	bad
medium	normal	u_5	u_4
	above		$\underline{u_3}$
high	normal		$\underline{u_8}\ \underline{u_{10}}$
	above	u_9	$u_6\ u_7$

Note that Pressure = low has been eliminated in the above because of Rule 1. There are two positive regions in the above, yielding two certain rules:

Rule 2: if Pressure = medium and Temp = above, then Quality = bad.
Rule 3: if Pressure = high and Temp = normal, then Quality = bad.

Similar analysis for (Pressure, Size) will give three rules:

Rule 4: if Pressure = medium and Size = fine, then Quality = good.
Rule 5: if Pressure = medium and Size = coarse, then Quality = bad.
Rule 6: if Pressure = high and Size = fine, then Quality = bad.

Similarly, from (Temp, Size) analysis, we have:

Rule 7: if Temp = above and Size = fine, then Quality = bad.

Cases of three condition attributes

After the above analyses, two elements, u_6 and u_9, remain. Both elements are in the same subset for Pressure = high, Temp = above, and Size = coarse, and Quality is bad for u_6 and good for u_9. Hence, we have an uncertain rule:

Rule 8: if Pressure = high, Temp = above, and Size = coarse, then Quality = good with 0.5 confidence factor and Quality = bad with 0.5 confidence factor.

We can also reduce Rule 8 to the following Rule 8' since Temp is superfluous.

Rule 8': if Pressure = high and Size = coarse, then Quality = good with 0.5

confidence factor and Quality = bad with 0.5 confidence factor.

6.8.2 ID3 Approach and the Case Study

ID3 is a machine learning technique based on constructing a decision tree from given data. There have been newer versions called C4.5 and C5, but the original name ID3 is still often used to refer to the family of this technique and we will follow this convention. As mentioned earlier, the decision tree approach is the most successful machine learning technique in traditional AI in terms of practical applications. (For practical applications of traditional machine learning, see P. Langley and H.A. Simon, Applications of Machine Learning and Rule Induction, *Communications of the ACM*, Vol. 38, No. 11 (Nov. 1995), 54-64.) ID3-type techniques are normally discussed in the traditional AI and machine learning literature, the counterpart of this book. In the following, we will start with a measure of uncertainty in probability theory, and the concepts of entropy in information theory. Entropy is employed by ID3 to measure information gain during the classification process of data. We will then briefly overview the ID3 approach and apply it to our case study problem. (For more on ID3, see J.R. Quinlan, *C4.5: Programs for Machine Learning*, Morgan Kaufmann, 1993.)

The basic idea of ID3 is classification of data by means of constructing a decision tree. For example, suppose that we are given data similar to Table 6, with the three condition attributes and one decision attribute. There may be 100 elements in the universe, u_1 to u_{100}. We determine which one of the three condition attributes best contributes to classifying the 100 elements into two groups of Quality = good and Quality = bad. To determine the "best" contribution, we use a measure called entropy. We compute the entropy for each of the three cases corresponding to the three condition attributes, then pick the best one. Perhaps Pressure is the best attribute. Then Pressure will be the first attribute to be used to classify the data, at the root of the decision tree. For the following levels of the tree, the same procedure is repeated recursively for further classification of the data. For example, at the branch for Pressure = low, we compute the entropy for each of the remaining attributes, Temp and Size, and pick the best one. We repeat this process for Pressure = medium and high. The meaning of these statements will be clearer when we see the following case study.

Probability distribution and a measure of uncertainty

A set of outcomes or events is called an outcome space. Each outcome is denoted by ω, and the outcome space by Ω. Any expression or statement that assigns to each outcome a real number is called a random variable, denoted as $X(\omega)$. We can determine probability distributions $P(\omega)$ and $P(X(\omega) = x)$. $P(\omega)$ is the probability that the outcome is ω. $P(X(\omega) = x)$ is the probability that the random variable X has the value of x. Often we forget about the underlying outcome space and work directly with the distribution of the random variable as $P(x)$.

Example. Tossing a coin twice

$\Omega = \{HH, HT, TH, TT\}$ and ω can be any one of HH, etc. Let a random variable $X(\omega)$

be the number of heads. Then $X(HH) = 2$, $X(HT) = 1$, $X(TH) = 1$ and $X(TT) = 0$. The probability distribution $P(\omega)$ is: $P(HH) = 0.25$, $P(HT) = 0.25$, $P(TH) = 0.25$, and $P(TT) = 0.25$. The probability distribution of the random variable X is: $P(X(\omega) = 2)$ or by forgetting the underlying outcome space, $P(2) = 0.25$; similarly, $P(1) = 0.5$, and $P(0) = 0.25$.

We define a measure of uncertainty or indetermination for a piece of information i as

$$-\log_2 P(i) \quad \text{bits},$$

where $P(i)$ is the probability corresponding to information i. This measure represents the number of bits required to describe the information. We often omit the base 2 of log.

Example. HH, etc. for tossing a coin twice

$-\log P(HH) = -\log (1/4) = -\log 2^{-2} = 2$ bits. Similarly, $-\log P(HT) = -\log P(TH) = -\log P(TT) = 2$ bits. These information can be represented by 2 bits as: $HH \rightarrow 11$, $HT \rightarrow 10$, $TH \rightarrow 01$ and $TT \rightarrow 00$.

Entropy in information theory

Entropy in information theory is originated from the concept of entropy in thermodynamics and statistical physics. In the latter, entropy represents the degree of disorder in a substance or a system. Similarly, entropy in information theory is a measure to represent the uncertainty of a message as an information source. The more information in a message, the smaller the value of the entropy.

Entropy $\mathcal{E}$ of a random variable $X(\omega)$ with a probability distribution $P(X(\omega) = x)$ is defined by

$$\mathcal{E} = -\sum_{\text{all } x} P(X(\omega) = x) \times \log_2 P(X(\omega) = x)$$

The entropy is a measure of the expected (that is, average) uncertainty or indetermination in the random variable. It is the number of bits on the average required to describe the random variable. We will use the convention that $0 \log 0 = 0$, since $\lim_{x \rightarrow 0} x \log x = 0$.

Example.

Consider a random variable which has a uniform probability distribution over 8 outcomes. $P(i) = 1/8$ for each $i = 1$ to 8.

$$\mathcal{E} = -\sum_{i=1}^{8} P(i) \times \log_2 P(i) = -\sum_{i=1}^{8} P(1/8) \times \log_2 P(1/8) = 3 \text{ bits}$$

To identify an outcome, we need a label that takes on 8 different values; 3-bit labels, 000, 001, ..., 111, suffice the requirement.

Basics of ID3

As described before, ID3 is a machine learning technique based on constructing a decision tree. The original ID3 used a criterion called *gain*. The gain is the amount of reduction in the entropy $\mathcal{E}$ when a data set is partitioned based on a certain parameter (attribute).

$$\text{gain} = (\mathcal{E} \text{ of before partitioning}) - (\mathcal{E} \text{ of after partitioning})$$

For some years the gain was used and gave good results. But it has a deficiency - it has a bias in favor of test with a parameter with many possible values. To correct this, we can use *gain ratio* defined as follows. First define *split info* as

$$\text{split info} = -\sum_i \left\{ \frac{|S_i|}{|U|} \times \log\left(\frac{|S_i|}{|U|} \right) \right\}$$

where U is the universe of elements before partitioning, and S_i's are subsets of U as the result of partitioning. The gain ratio is then defined as

$$\text{gain ratio} = \frac{\text{gain}}{\text{spilt info}}$$

ID3 applied to case study

We will apply the ID3 approach using the gain criterion to the case study problem presented in the previous section. At the end, we will briefly show computation of gain ratio. The universe U contains 10 elements, $\{u_1\ u_2, u_3, u_4, u_5, u_6, u_7, u_8, u_9, u_{10}\}$.

Entropy $\mathcal{E}(0)$ before any partitioning

Out of the 10 elements in the universe, four elements, $u_1\ u_2$, u_4, and u_9, are Quality = good, and the remaining six elements are Quality = poor. The probability of Quality = good is 4/10, and the probability of Quality = poor is 6/10. Hence, the entropy of random variable Quality = good or bad is

$$\begin{aligned}\mathcal{E}(0) &= -[P(\text{Quality = good}) \times \log P(\text{Quality = good}) \\ &\quad + P(\text{Quality = bad}) \times \log P(\text{Quality = bad})] \\ &= -\left[\frac{4}{10} \times \log\left(\frac{4}{10} \right) + \frac{6}{10} \times \log\left(\frac{6}{10} \right) \right] = 0.529 + 0.442 = 0.971\end{aligned}$$

Entropy $\mathcal{E}$ at level-1 partitioning

Partitioning by Temp

When the universe is partitioned by attribute Temp, we have two subsets, corresponding to Temp = normal and Temp = above, as is shown in Fig. 6.13.

We compute the entropy of each subset, then the weighted sum of these two entropies as the total entropy for the partitioning the universe by Temp. In the subset (that is, the branch in the decision tree) for Temp = normal, there are six elements, u_1 u_2, u_4, u_5, u_8, and u_{10}. Of these, Quality = good for three elements, u_1 u_2, and u_5, and Quality = bad for three elements, u_4, u_8, and u_{10}. The entropy of the Temp = normal branch is -[(3/6) log (3/6) + (3/6) log (3/6)] = -[(1/2) × (-1) + (1/2) × (-1)] = 1. Similarly, we can compute the entropy of the Temp = above branch of u_3, u_6, u_7, u_9,

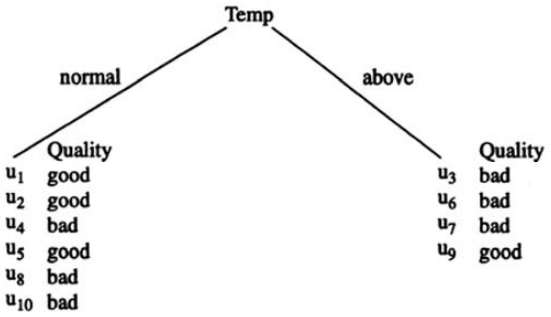


Fig. 6.13. A decision tree partitioned by Temp.

as -[(1/4) log (1/4) + (3/4) log (3/4)] = 0.25 × 2 + 0.75 × 0.4150 = 0.811. We take the weighted sum of these two entropies as the entropy of partitioning U by Temp. The weight of each branch is the ratio of the number of elements in the branch over the total number of elements in the universe.

$$\mathcal{E}(\text{Temp}) = \left(\frac{6}{10}\right) \times 1 + \left(\frac{4}{10}\right) \times 0.811 = 0.925.$$

The gain by partitioning by Temp then is computed by

$$\mathcal{E}(0) - \mathcal{E}(\text{Temp}) = 0.971 - 0.925 = 0.046$$

Partitioning by Pressure

Similarly, we can compute $\mathcal{E}$(Pres), the entropy of partitioning U by Pressure. In this case, we have three branches, corresponding to Pres = low, medium and high, as is shown in Fig. 6.14.

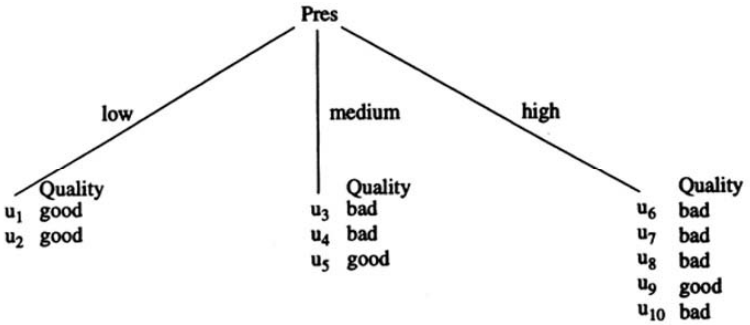


Fig. 6.14. A decision tree partitioned by Pressure.

We compute the entropy of each branch as follows. For Pres = low, -[(2/2) log (2/2)] = 0; for Pres = medium, -[(1/3) log (1/3) + (2/3) log (2/3)] = 0.918; for Pres = high, -[(1/5) log (1/5) + (4/5) log (4/5)] = = 0.722. $\mathcal{E}$(Pres) as the weighted sum of these entropies is

$$\mathcal{E}(\text{Pres}) = \left(\frac{2}{10}\right)\times 0 + \left(\frac{3}{10}\right)\times 0.918 + \left(\frac{5}{10}\right) \times 0.722 = 0.636 .$$

The gain by partitioning by Pres is

$$\mathcal{E}(0) - \mathcal{E}(\text{Pres}) = 0.971 - 0.636 = 0.335$$

This gain is larger than the gain by partitioning by Temp. The larger the gain, the better.

Partitioning by Size

Similar computation yields $\mathcal{E}$(Size) = 0.925, hence

$$\mathcal{E}(0) - \mathcal{E}(\text{Size}) = 0.971 - 0.925 = 0.046$$

Conclusions at level-1 partitioning. The gain by partitioning by Pressure is the highest, and we choose Pressure to be the attribute to partition the universe as in Fig. 6.14. In the Pres = low branch of Fig. 6.14, both elements, u_1 and u_2, have Quality = good. The entropy of this branch is zero, and it leads to a rule

Rule 1: if Pressure = low, then Quality = good

Entropy $\mathcal{E}$ at level-2 partitioning

For the remaining two branches (subtrees) in Fig. 6.14, corresponding to Pres = medium and high, their entropies are not zero, and we need further partitioning by the remaining attributes, Temp and Size. Computation of entropies of the branches is the

same as before, that is, the same scheme is applied recursively to smaller subtrees.

Pres = medium subtree

Before further partitioning, this subtree has the entropy of $\mathcal{E}$(Pres = medium) = 0.918. When this subtree is further partitioned by Temp, the entropy $\mathcal{E}$(Pres = medium, Temp) = (2/3) × {-[(1/2) log (1/2) + (1/2) log (1/2)]} + (1/3) × {-[(1) log (1)]} = (2/3) × 1 + (1/3) × 0 = 0.667. The gain is 0.918 - 0.667 = 0.251. When partitioned by Size, the entropy $\mathcal{E}$(Pres = medium, Size) = (1/3) × 0 + (2/3) × 0 = 0. The gain is 0.918 - 0 = 0.918. Hence, we choose Size for further partitioning for this Pres = medium subtree.

Pres = high subtree

Before further partitioning, this subtree has the entropy of $\mathcal{E}$(Pres = high) = 0.722. When this subtree is further partitioned by Temp, the entropy $\mathcal{E}$(Pres = high, Temp) = 0.551. The gain is 0.722 - 0.551 = 0.171. When partitioned by Size, the entropy $\mathcal{E}$(Pres = high, Size) = 0.4. The gain is 0.722 - 0.4 = 0.322. Hence, we choose Size for further partitioning for this Pres = high subtree.

The result of level-2 partitioning is shown in Fig. 6.15. We see three new branches whose entropies are zero. Corresponding to these three branches, we can have three rules.

Rule 2: if Pressure = medium and Size = fine, then Quality = good.
Rule 3: if Pressure = medium and Size = coarse, then Quality = bad.
Rule 4: if Pressure = high and Size = fine, then Quality = bad.

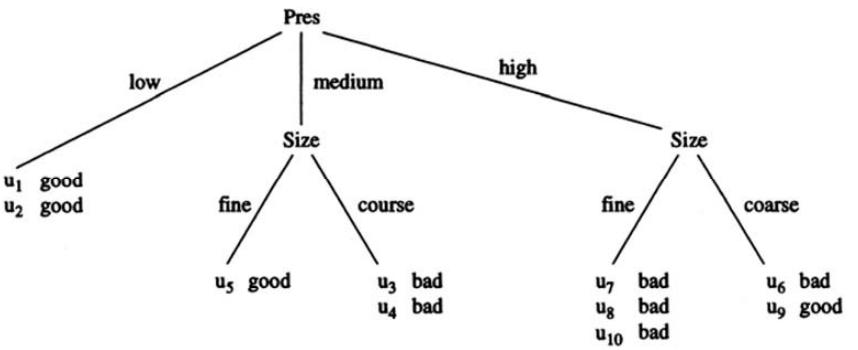


Fig. 6.15. A decision tree after level-2 partitioning.

Entropy $\mathcal{E}$ at level-3 partitioning

At the branch of Pressure = high and Size = coarse, we need further partitioning by Temp to gain more information. However, in this example, both elements, u_6 and u_9,

have Temp = above, thus additional partitioning by Temp does not reduce the entropy. Therefore, we stop our process here. From this last branch we may say

Rule 5: if Pressure = high and Size = coarse, then Quality = good with 0.5 and bad with 0.5 confidence factor and bad with 0.5 confidence factor.

Gain ratios

Although we have used the gain criterion for simplicity in the preceding discussions, an alternative is the gain ratio as discussed earlier. The gain ratio for partitioning by Temp at level-1 can be computed as follows.

split info = -[(6/10) log (6/10) + (4/10) log (4/10)] = 0.971

gain ratio = gain / split info = 0.046 / 0.971 = 0.047

The gain ratio for partitioning by Pressure at level-1 can be computed as follows.

split info = -[(2/10) log (2/10) + (3/10) log (3/10) + (5/10) log (5/10)] = 1.4855

gain ratio = gain / split info = 0.521 / 1.4855 = 0.351

Similarly, the gain ratio for partitioning by Size can be computed as 0.047. In our case study, the gain ratio criterion will give the same result as using the gain, that is, partitioning by Pressure at level-1. Generally, these two criteria can result in different partitioning. When an attribute has many possible values, such as very low, moderately low, slightly low, and so forth, partitioning by this attribute tends to yield a low entropy value since each branch will have a relatively small number of elements. If another attribute has fewer possible values, say, only two as low and high, and still gives the same entropy value as using an attribute with many possible values, this new attribute is much better for classification. This is why the gain ratio may be a better criterion than the gain.

Comparison of rule inductions by rough sets and ID3

After studying rule induction examples by rough set theory and ID3, we may wonder how these two approaches can be compared. For one, we note that rough sets are a mathematical theory, as set theory. On the other hand, ID3 is a machine learning technique for rule induction. Hence, it may not be appropriate to compare these two on the same ground. Nevertheless, both can be used for rule induction as we have just seen, and we would like to make some comparisons on this aspect. Comparisons in general terms, however, are difficult because there are so many different cases. In the following, we present rule-of-thumb guidelines which may apply in many situations.

Both approaches typically work on a table form of raw data like Table 6. In our case study, rough set theory and ID3 give somewhat similar results. In general, rough set theory and ID3 may give similar results for relatively small problems. For others, the results can be quite different. How to efficiently derive rules, and in what form, are the key problems in machine learning.

Obviously these two methods employ different classification criteria. Rough set theory is typically based on the relations between the condition and decision attributes, such as positive and boundary regions, and reducts and cores. ID3 uses entropy as a measure for its classification process. One might argue that ID3 is based on entropy, which is a concrete quantitative measurement in information theory, but it might require more computation time. Each method, however, can be modified or extended. Depending on the modification or extension, these methods can be closer. For example, we can compute entropy in rough sets as a measure of certainty for uncertain rules.

More importantly, the induced knowledge from these methods can be different. For our case study, Rules 2, 3 and 7 derived by rough sets are not by ID3. One might say that the rules derived from rough sets are more extensive, while ID3 focuses on important rules based on the entropy criterion. If there are too many rules to deal with, ID3 would be more efficient. However, ID3 could overlook potentially useful rules. Again, these methods can be modified or extended. For example, less important rules derived from rough sets can be pruned based on a some measure.

Also, the ways to represent derived knowledge or rules in these two methods are different. Rough set theory is based on non-tree structure, typically two-dimensional tables, while ID3 is based on decision trees. Not every knowledge would be best represented by either a table or a tree. Certain classes of problems are probably best represented by tables, some by trees, and still others by some different data structures. Searching for specific rules in a knowledge base of tree form is generally efficient. We recall that many efficient algorithms are based on tree structures, such as binary search and heapsort. One might argue that merging trees for knowledge base restructuring could be harder than merging tables.

In a more general perspective, both techniques described here, which are based on raw data like Table 6, are called attribute-based learning. The idea is that we have sample data for a set of condition and decision attributes, and our objective is to extract knowledge from the data. Certainly this would not be the only way to present data. Major advantages of attribute-based learning are its relative simplicity, efficiency, and a capability of handling noisy data. Its disadvantages are limited capability of expressing the underlying knowledge, and lack of specifying relations among parts of the elements.

All of these advantages and disadvantages discussed above are debatable issues. As said before, there is no mathematical proof to answer these questions for general cases. Probably a consensus would be that no single approach is the best for all problems. That is, ID3 would be more effective for certain classes of problems, while rough sets for others.

6.8.3 Comparisons with Other Techniques

How rough set theory differs from statistical methods

Both approaches deal with similar problems, i.e., reasoning about data. They are different, however, and certain problems are solved better by traditional statistical methods while the other by rough sets, and they can complement each other. For example, in statistical analysis, the target data under investigation is assumed to be multivariate normal distribution. In the rough set approach, no such assumption is

made. This means that when there is a large amount of sampling data and its distribution is close to normal, the classical statistics may work better; otherwise, i.e., either the sampling is too small and/or there is a non-normal distribution, rough sets may perform better. But this is only a guideline; there is no theory that predicts exactly under what circumstance which approach works better.

In discriminant analysis, we compute statistical characteristics such as the means and covariance matrices, then we classify the data objects to one of these classes on the basis of the description with the discriminant scores. The result can be viewed as an algorithm in the form of functional representation. In rough sets, we do not compute means, and so on, but work directly on the data. We reduce redundant attributes, arriving at minimal subsets of attributes ensuring the same quality of classification. The result is an algorithm composed of logical decision rules. (For more see Stefanowski, 1992.)

Examples for which the statistical method failed but rough sets succeeded include the following: airline pilot performance evaluation, geographical data classification, questionnaire analysis in sociology and psychology, and many medical applications (Pawlak, et.al., 1988).

Dempster-Shafer and rough set theories

The Dempster-Shafer theory of evidence, is a relatively recent technique especially for dealing with uncertain knowledge. The major difference between the Dempster-Shafer and rough set theories is that the former uses belief functions as a main tool, while the latter makes use of relations among the attributes such as the lower and upper approximation sets. Although the Dempster-Shafer theory can be employed for induction, its major objectives are dealing with uncertain knowledge and approximate reasoning. The Dempster-Shafer theory can be said to be subjective (based on an expert's judgment) while rough set theory is objective (based on the data).

Crisp, fuzzy, and rough sets: an illustration

Consider a group of people, such as students in a class, an audience at a conference, or employees in a department of a company, as the universe under consideration. The set of women of this universe is a crisp set; there is no vagueness involved for this set. The set of young people is a good example of a fuzzy set. Since the measure of youngness does not change abruptly from 1 to 0 at a certain age, say 30, it is natural to associate a degree of youngness to each person. A 25 years old person has a degree of 1, 30 years 0.9, 35 years 0.5, etc.

In this fuzzy set example, we assume the age of each person is known. Let us imagine a hypothetical experiment where the age is not known and we want to estimate the age from observation. There are many condition attributes to be observed, such as facial appearance, whether the hair is grey, white or the head is bald, how the voice sounds, and so on. We record the value of each attribute for each person to the best of our ability whenever it is known. We then come up with an estimated age for each person, which is the decision attribute of this problem. This is a rough set problem.

Suppose that we automate this process using a TV camera, a microphone, and of

course a computer. Thanks to machine vision and speech processing technologies, the machines can record the condition attribute values automatically. We may use 1,000 people, whose ages are already known, as training samples. The raw data are crude, some are incomplete or inaccurate, and some are irrelevant. Rough sets will tell which condition attributes are important, and to what degree, and how their values can be used to determine the values of the decision attribute - the age. (Similarly, humans are also able to accumulate such experience over years to build common sense and to estimate a person's age from observed information.) Perhaps the color of clothes a person wears is not important and may be dropped from the condition attributes. Sometimes, we can intentionally drop some condition attributes and see how the system performs - weighing a trade-off between the amount of work involved for data gathering and subsequent analysis, and the accuracy of the resulting age estimate. After this training session, we perform our original experiment on the group of people, students in a class, or whatever else. Rough sets will tell us the best estimates of their ages under the given circumstance.

We might wonder whether fuzzy set theory can achieve the same, i.e., estimating ages of people from observed data. Yes, it can, but it would be much more time-consuming. What we have to do is to develop many fuzzy if-then rules, such as "if a person's hair is ... then ...." As we can see in this example, rough and fuzzy set theories can complement to each other. For example, in the above age group problem, rough sets can be used as a front-end of a fuzzy system. Conversely, for certain problems a fuzzy system can be used as a front-end or an aid of a rough set system. For example, some condition attribute values may be expressed in terms of fuzziness, which may be dealt with fuzzy logic. Some condition attribute values may be pre-processed using fuzzy if-then rules yielding a smaller number of intermediate attribute values. Mapping from condition to decision attributes may involve fuzziness, etc. Or, rough and fuzzy set subsystems can simply coexist to derive and exchange information back and forth between them.

Further Reading

Z. Pawlak, “Rough Sets,” *International Journal of Computer and Information Science*, 11, 341-356, 1982 (a seminal article).

Z. Pawlak, *Rough Sets: Theoretical Aspects and Reasoning about Data*, Dordrecht, Netherlands: Kluwer Academic, 1991 (a comprehensive book).

Z. Pawlak, J. Grzymala-Busse, R. Slowinski, and W. Ziarko, “Rough Sets,” *Communications of the ACM,* Vol. 38, No. 11, 89-95, Nov. 1995.

L. Polkowski, *Rough Sets:Mathematical Foundations*. Physica-Verlag, Heidelberg, 2002.

Z. Pawlak, “A Treatise on Rough Sets”, *Transactions on Rough Sets IV, Lecture Notes in Computer Science* (*LNCS*), Vol. 3700, Springer-Verlag, Berlin, 1-17, 2005.

R.Slowinski, S.Greco, B.Matarazzo: “Rough Set Based Decision Support,” Chapter 16, in E.K. Burke and G. Kendall (eds.), *Search Methodologies*: *Introductory*

Tutorials in Optimization and Decision Support Techniques, Springer-Verlag, New York, 475-527, 2005.

A journal that carries articles on rough sets.

Transactions on Rough Sets, Lecture Notes in Computer Science (*LNCS*), Springer, Berlin.

7 Chaos

7.1 What is Chaos?

In our daily life, one may say "that was complete chaos" to describe a situation of extreme disorder, irregularity, or confusion. We may wonder what chaos has to do with anything useful, particularly with intelligent computing. The scientific meaning of the term **chaotic system** or **chaos** for short, which we use in this chapter, has one distinctive characteristic among others. That is, chaos is a phenomena that has *deterministic underlying rules* behind irregular appearances.

An over-simplified view of chaotic and other phenomena in this world

For easy understanding, we may over-simplify the classification of phenomena into the following categories in terms of two parameters: 1) the regularity of the appearance on the surface; 2) the characteristics of the underlying rules.

Type of Phenomenon	Appearance	Underlying Rules
1. Regular	Regular	Deterministic
2. Statistical	Regular	Probabilistic
3. Chaos	Irregular	Deterministic
4. Random	Irregular	Probabilistic

Examples of these phenomena are:

1. Regular: The motion of a pendulum of a grandfather's clock. If it is not regular, it cannot serve as a device that keeps track of time.
2. Statistical: This is a type of phenomenon that looks regular on the surface, but is a statistical result of many probabilistic events behind the scene. For example, macroscopic motion of air flow may be regular, although microscopic motions of the molecules in the air may be probabilistic.
3. Chaos: A pseudo-random number generator. More examples will be discussed in this chapter.
4. Random: A sequence of lottery numbers. If there were any underlying rules, one could increase odds of winning the lottery.

Note that generally the boundary between deterministic chaos and probabilistic random systems may not always be clear since seemingly random systems could have deterministic underlying rules not yet found (perhaps only God knows at present).

Typical features of chaos

As with many terms in science, there is no standard definition of chaos. The typical, commonly-accepted features of chaos include:

1. *Deterministic*. It has deterministic rather than probabilistic underlying rules which every future state of the system must follow.

2. *Nonlinear*. The underlying rules are nonlinear; if they are linear, it cannot be chaos.

3. *Irregular*. The behavior of the system shows sustained irregularity. Hidden order includes a large or infinite number of unstable periodic patterns or motions. This hidden order forms the infrastructure of irregular chaotic systems -"order in disorder" in short.

4. *Sensitive to initial conditions*. Small changes in the initial state of chaotic systems can lead to radically different behavior in the final state. This "Butterfly Effect," presents the possibility that even a slight perturbation of a butterfly flapping its wings can dramatically affect whether sunny or cloudy skies will dominate days later.

5. Long term prediction is practically impossible in most cases due to sustained irregularity and sensitivity to initial conditions, which can only be known to a finite precision.

We will discuss a more mathematical description of chaos in Section 7.5.

Generally, nonlinear problems are difficult to solve analytically, except in a limited number of cases. This is why many scientists and engineers have shied away from nonlinear problems in general. Chaos can be said to be one of the hardest nonlinear problems. Historically, the study of chaos started in mathematics and physics. It then expanded into engineering, and more recently into information and social sciences. Recently, there has been growing interest in the theoretical study and applications of chaotic systems. There are several reasons for this recent interest in chaos research. One is the importance of the study in many disciplines. Another key element is the power, speed and memory capacity of easily accessible computer hardware. Although the history of chaotic systems research is not new, it was the computer revolution that gave life to their practical applications.

Simple examples of chaos

In the following, we will see some examples to get a feeling of chaotic systems.

Example 1. Pseudo-random number generator

A simple example of a chaotic system in computer science is a pseudo-random number generator. The underlying rule in this case is typically a simple deterministic formula. The resulting solutions, i.e., the pseudo-random numbers are, however, very irregular and unpredictable (the more unpredictable, the better the random numbers). We also note that a small change in the initial condition (seed) can yield a significantly different sequence of random numbers. These random number generators are chaotic but also periodic with certain periods. Such generators viewed carefully yield the hidden order characteristic of chaos.

To see chaotic behavior, let us use a simple generator of the form: $x_{t+1} = cx_t \bmod m$. If this equation is $x_{t+1} = cx_t$, it would be linear, but taking the modulus makes it nonlinear. For example, we can choose $c = 16807$ and $m = 2147483647$. (For more on pseudo-random number generators, see D.E. Knuth, The Art of Computer Programming, Vol. 2, *Seminumerical Algorithms*, 3rd Ed., 1997.)

For illustration purposes, we will use $x_{t+1} = 29x_t \bmod 997$. Fig. 7.1 shows two sequences of numbers for this generator, where one sequence has a seed of 117 while the other uses a slightly different value of 118. We note that although only the discrete corner points of the graph are meaningful, it is a common practice to connect these points with line segments for easier recognition of the sequences.

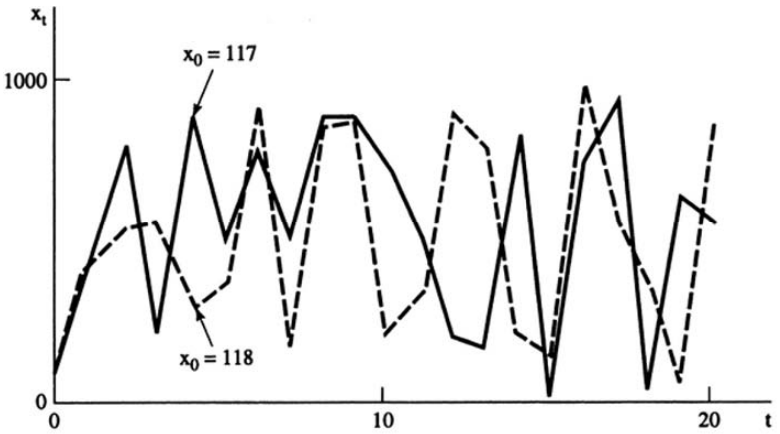


Fig. 7.1 Chaotic behavior for pseudo-random numbers.

Example 2. $x_{t+1} = 2x_t^2 - 1$

When we start with x_0 where $0 < x_0 < 1$, then x_t will be confined between -1 and 1. This simple nonlinear recurrence equation can demonstrate a chaotic behavior. Fig. 7.2 shows two sequences with $x_0 = 0.25$ and $x_0 = 0.26$.

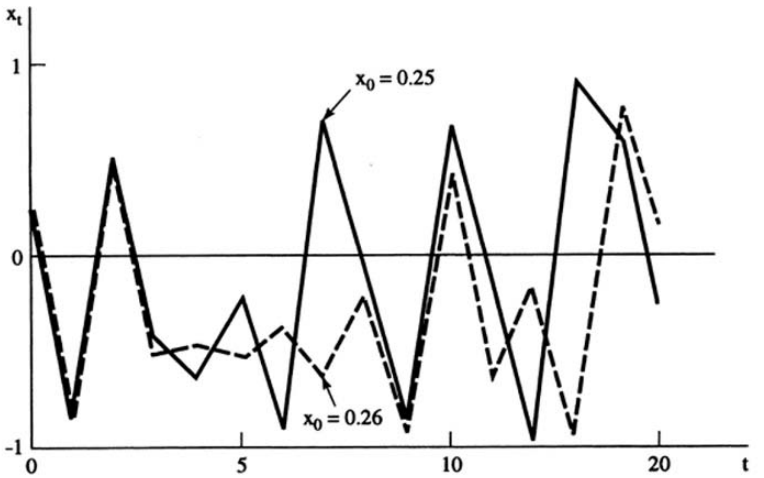


Fig. 7.2 Chaotic behavior for $x_{t+1} = 2x_t^2 - 1$, where $x_0 = 0.25$ and 0.26.

Other examples.

While the above two examples are man-made chaotic systems, there are numerous chaotic systems in nature. For example, normal brain activity may normally be chaotic and pathological order might indeed be the cause of diseases such as epilepsy. It has been speculated that too much periodicity in heart rates might indicate disease. Perhaps the chaotic characteristics of the human body are better adapted to its chaotic environment. It is even suspected that biological systems exploit chaos to store, encode and decode information.

Generally, for either a chaotic or non-chaotic system, a relationship between a dependent variable x_t vs. the independent variable time t, plotted on a diagram like Fig. 7.1 or Fig. 7.2, is called a **time series**. When x is a continuous function of t, such a diagram is called a **time waveform** or simply **waveform**. Analysis of time series or waveform is of interest in many disciplines such as engineering, natural science, medicine, and social science.

7.2 Representing Dynamical Systems

In this section we discuss basic mathematical concepts used to represent and analyze dynamical systems in general, for both chaotic and nonchaotic unless otherwise specified. A **dynamical system** is a system that changes its state over time *t*. Thereare two categories of dynamical systems: **discrete** and **continuous**. Many of the concepts for discrete and continuous systems apply to both categories.

7.2.1 Discrete dynamical systems

The two examples in the previous section, $x_{t+1} = cx_t \bmod m$ and $x_{t+1} = 2x_t^2 - 1$, are both discrete dynamical systems. A discrete dynamical system describes the changes of the system for discrete values of *t*, for example, for $t = 0, 1, 2, \ldots$. An equation for a discrete dynamical system such as, $x_{t+1} = cx_t \bmod m$ or $x_{t+1} = 2x_t^2 - 1$, can generally be represented as follows:

$$x_{t+1} = f(x_t)$$

where $f(x_t)$ is some function of x_t. This equation is called by various names such as a **recurrence equation**, **recurrence relation**, **difference equation**, or a **map** - since the equation maps the state at time *t* to the state at time $t + 1$. Time *t* is the *independent variable* since its value does not depend on any other values; *x* in the above examples is the *dependent variable*.

The above recurrence equation can further be generalized as $x_{t+1} = f(x_t, t)$; that is, function *f* explicitly includes the independent variable *t*. Such a recurrence equation is called **nonautonomous**. For example, $x_{t+1} = tx_t^2 - 1$ is an example. On the other hand, the earlier form of a recurrence equation $x_{t+1} = f(x_t)$, that is, *f* is a function of only x_t and not explicitly of *t*, is called **autonomous**. We will consider autonomous and nonautonomous systems in the next subsection for continuous dynamical systems, but we will mostly discuss autonomous systems in this chapter.

Any function $x_t = \varphi(t)$ that identically satisfies the recurrence equation $x_{t+1} = f(x_t)$ is called a **solution**. A solution that satisfies the recurrence equation together with a specific initial condition, say, $x = x_0$ at $t = 0$, is called a **particular solution**. A solution $x = \varphi(t)$ that identically satisfies the recurrence equation without specific initial condition is called a **general solution**, to distinguish it from a particular solution.

Steady state refers to the asymptotic behavior of the solution as time approaches infinity. A system starts with an initial condition at time $t = 0$, progresses through the **transient states**, then may reach its steady state. We note that when we say that time approaches infinity, we tend to think the time is very long, for example, a billion years or more. However, usually this is not the case. For example, when the transient states are in the order of milli-seconds, the system will reach its steady state within a fraction of a second That is, infinity in this case means a fraction of a second.

Multi-dimensional discrete dynamical systems

The two examples for Figs. 7.1 and 7.2 are *one-dimensional*, since there is only one dependent variable x. When a system is described by two dependent variables, x and y, it is a *two-dimensional discrete dynamical system*. A system of two simultaneous recurrence equations, $x_{t+1} = 2x_t^2 - 1$, $y_{t+1} = x_t y_t$, is an example. A three-dimensional discrete dynamical system has three dependent variables, x, y, and z. We can further extend these to an *n-dimensional discrete dynamical system*, having n dependent variables, x_1, x_2, ..., x_n, or $x^{(1)}$, $x^{(2)}$, ..., $x^{(n)}$. (Hereafter, we will use x_1, ..., rather than $x^{(1)}$,..., notation. Note that for a general n or often for $n \geq 4$, these subscripted or superscripted expressions are used since the number of letters is limited. On the other hand, for up to $n = 3$, x, y, and z are often used since they are easier to understand than, e.g., x_1, x_2, and x_3.)

By extending a one-dimensional system, $x_{t+1} = f(x_t)$, to an n-dimensional system, we have:

$$x_{1,\,t+1} = f_1(x_{1,\,t}, x_{2,\,t}, ..., x_{n,\,t})$$

$$x_{2,\,t+1} = f_2(x_{1,\,t}, x_{2,\,t}, ..., x_{n,\,t})$$

. . .

$$x_{n,\,t+1} = f_n(x_{1,\,t}, x_{2,\,t}, ..., x_{n,\,t})$$

This set of simultaneous recurrence equations can be represented more compactly by using two vectors, $\mathbf{x}_t = (x_{1,\,t}, x_{2,\,t}, ..., x_{n,\,t})$ and $\mathbf{f}(\mathbf{x}_t) = (f_1(\mathbf{x}_t), f_2(\mathbf{x}_t), ..., f_n(\mathbf{x}_t),)$, as,

$$\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t)$$

Linearity and nonlinearity

Since chaos is a science of nonlinear systems, we need to clearly understand what is linear and what is not. A recurrence equation is **linear** if each term is in form of $\alpha(t) \times x$, where $\alpha(t)$ is any function of independent variable t, including a constant, and x is a *dependent variable*. The dependent variable x can be x_t, y_t, $x_{1,\,t}$, $x_{2,\,t}$, etc. In words, a recurrence equation is linear if each term is one dependent variable to its first power, possibly multiplied by a constant factor or by any function of the independent variable. All others are **nonlinear**. Common forms of nonlinear include: a power of polynomial of x, where the power can be real or negative, e.g., x^2, $x^{0.5} = \sqrt{x}$, $x^{-1} = 1/x$; a function of x, such as $\sin x$, $\log x$, e^x and taking modulus; or a "cross term," such as xy, for multi-dimensional systems. Here are some examples of linear and nonlinear systems:

Linear examples.

1. $x_{t+1} = e^{-t}x_t - \sin t$

2. $x_{t+1} = 2x_t - 1$

$y_{t+1} = x_t - y_t$

Nonlinear examples.

1. $x_{t+1} = x_t^2 - 1$

2. $x_{t+1} = 2 - 1/x_t$
   $y_{t+1} = x_t y_t$

7.2.2 Continuous dynamical systems

A *continuous dynamical system* describes the dynamic nature of the system for continuous values of t, using the time derivatives of the dependent variables. The dependent variables can be represented as x, y, z, or x_1, x_2, ..., x_n. When we want to explicitly indicate these dependent variables are functions of time t, we can write them as: $x(t)$, $y(t)$, $z(t)$; x_t, y_t, z_t; $x_1(t)$, $x_2(t)$, ..., $x_n(t)$; or $x_{1,t}$, $x_{2,t}$, ..., $x_{n,t}$. We will represent the first and second time derivatives by overdots as $dx/dt = \dot{x}$ and $d^2x/dt^2 = \ddot{x}$, and the higher time derivatives in form of $d^nx/dt^n = x^{(n)}$. Here are some examples of continuous dynamical systems:

1. $\dot{x}$ = sin t first-order, one-dimensional
2. $\ddot{x}$ + a $\dot{x}$ + sin x = 0second-order, one-dimensional
3. $\dot{x}$ = y
   $\dot{y}$ = - ay - sin x first-order, two-dimensional

As in the case of discrete systems, time t is the independent variable and x, y, (and z, so on, for higher dimensions) represent the dependent variables. From study of differential equations, we recall that the **order** of a differential equation is the order of the highest derivative which occurs. In the second example, the highest order derivative is $\ddot{x}$, which is the second order.

In general, a first-order, one-dimensional continuous dynamical system can be described by:

$$\dot{x} = f(x, t)$$

where $f(x, t)$ is a some function of x and t.

The notes for solution and steady state in the previous subsection for discrete systems also apply to continuous dynamical systems. That is, any function $x = \varphi(t)$ that identically satisfies the differential equation $\dot{x} = f(x, t)$ is called a **solution**. A solution that satisfies the differential equation together with a specific initial condition, say, $x = x_0$ at $t = 0$, is called a **particular solution**. A solution $x = \varphi(t)$ that identically satisfies the differential equation without specific initial condition is called a **general solution**, to distinguish it from a particular solution. **Steady state** refers to the asymptotic behavior of the solution as time approaches infinity. A system starts with an initial condition at time $t = 0$, goes through the **transient states**, then may reach its steady state.

Multi-dimensional continuous dynamical systems

A first-order, two-dimensional system is an extension of the one-dimensional as,

$$\dot{x} = f_1(x, y, t)$$
$$\dot{y} = f_2(x, y, t)$$

For a further extension to an n-dimensional system, we have,

$$\dot{x}_1 = f_1(x_1, x_2, ..., x_n, t)$$
$$\dot{x}_2 = f_2(x_1, x_2, ..., x_n, t)$$
$$\cdots$$
$$\dot{x}_n = f_n(x_1, x_2, ..., x_n, t)$$

This set of simultaneous differential equations can be represented more compactly by using two vectors, $\mathbf{x} = (x_1, x_2, ..., x_n)$ and $\mathbf{f}(\mathbf{x}, t) = (f_1(\mathbf{x}, t), f_2(\mathbf{x}, t), ..., f_n(\mathbf{x}, t))$, as,

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$$

where $\dot{\mathbf{x}} = (\dot{x}_1, \dot{x}_2, ..., \dot{x}_n)$. Any function $\mathbf{x} = \boldsymbol{\varphi}(t) = (\varphi_1(t), \varphi_2(t), ..., \varphi_n(t))$ that identically satisfies the differential equation $= \mathbf{f}(\mathbf{x}, t)$ is called a solution. Other concepts, such as particular and general solutions and steady state, can also be extended similarly to higher dimensions.

Linearity and nonlinearity

The notes for linearity and nonlinearity for discrete systems also apply to continuous systems, with one change - "a dependent variable" in discrete case is replaced by "a dependent variable or its (any order) derivative." That is, a system of differential equations is **linear** if each term is in the form of $\alpha(t) \times x$, $\alpha(t) \times \dot{x}$, $\alpha(t) \times \ddot{x}$, and so on, where $\alpha(t)$ is any function of independent variable t, including a constant, and x is a *dependent variable*. All others are **nonlinear**. Here are some examples of linear and nonlinear systems:

Linear examples.

1. $\dot{x} = x - \sin t$
2. $\ddot{x} + a\dot{x} + \sin^2 t \cdot x = b$
3. $\dot{x} = y$
   $\dot{y} = -ay - x$

Nonlinear examples.

1. $\dot{x} = x^2 - \sin t$
2. $\ddot{x} + a\dot{x} + \sin x = 0$

3. $\dot{x} = y$
   $\dot{y} = -axy$

Reducing a high-order differential equation to a set of first-order differential equations

We note that an *m*-th order, one-dimensional dynamical system can always be reduced to a system of first-order, *m*-dimensional differential equations by introducing new dependent variables. For example, a second-order, one-dimensional system: $\ddot{x} + a\dot{x} + \sin x = 0$ can be reduced to first-order, two-dimensional: $\dot{x} = y$ and $\dot{y} = -ay - \sin x$ by introducing a new dependent variable *y*. More generally, an *m*-th order, one-dimensional dynamical system can be described by,

$$x^{(m)} = f(x, , , ..., x^{(m-1)}, t)$$

We introduce new dependent variables as: $y_1 = x, y_2 = \dot{x}$, $y_3 = \ddot{x}$, ..., $y_m = x^{(m-1)}$. Then the reduced set of first-order, *m*-dimensional differential equations are:

$$\dot{y}_1 = y_2$$
$$\dot{y}_2 = y_3$$
$$\cdots$$
$$\dot{y}_{m-1} = y_m$$

(and from the original equation)

$$\dot{y}_m = f(y_1, y_2, ..., y_m, t)$$

This technique can further be extended for reducing a set of high-order, multi-dimensional differential equations to a system of first-order, multi-dimensional differential equations by introducing new dependent variables for each of the original variables.

Autonomous and nonautonomous dynamical systems

A dynamical system $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$, that is, **f** is a function of only **x** and not explicitly of *t*, is called **autonomous**; otherwise, that is $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, t)$, **nonautonomous**. Note that $\mathbf{f}(\mathbf{x})$ of an autonomous system is still implicitly a function of *t*, since $\mathbf{f}(\mathbf{x})$ is a function of **x**, and in turn **x** is a function of *t*.

We are often interested in a special case of a nonautonomous dynamical system called **time-periodic**, where **f** is a cyclic function in terms of *t*, that is, $\mathbf{f}(\mathbf{x}, t + T) = \mathbf{f}(\mathbf{x}, t)$ for $T > 0$. Note that if $\mathbf{f}(\mathbf{x}, t + T) = \mathbf{f}(\mathbf{x}, t)$, then $\mathbf{f}(\mathbf{x}, t + 2T) = \mathbf{f}(\mathbf{x}, t)$ since $\mathbf{f}(\mathbf{x}, t + 2T) = \mathbf{f}(\mathbf{x}, (t + T) + T) = \mathbf{f}(\mathbf{x}, (t + T)) = \mathbf{f}(\mathbf{x}, t)$. Similarly, $\mathbf{f}(\mathbf{x}, t + 3T) = \mathbf{f}(\mathbf{x}, t)$, $\mathbf{f}(\mathbf{x}, t + 4T) = \mathbf{f}(\mathbf{x}, t)$, and so on. The smallest such *T* is called the **minimal period**. A time-periodic nonautonomous system can always be reduced to an autonomous system of one additional dimension by employing a trick as follows. Define a new dependent variable x_{n+1} as:

$$x_{n+1} = (2\pi/T)t \text{ or equivalently } t = (T/2\pi)x_{n+1}$$

Then the original set of equations reduces to:

$$\dot{x}_1 = f_1(x_1, x_2, ..., x_n, \frac{T}{2\pi} x_{n+1})$$
$$\dot{x}_2 = f_2(x_1, x_2, ..., x_n, \frac{T}{2\pi} x_{n+1})$$
$$\cdots$$
$$\dot{x}_n = f_n(x_1, x_2, ..., x_n, \frac{T}{2\pi} x_{n+1})$$
$$\dot{x}_{n+1} = \frac{2\pi}{T}$$

Or, using vector $\mathbf{x} = (x_1, x_2, ..., x_n)$, this set can be represented more compactly as (Note that $\mathbf{x}$ is a vector while x_{n+1} is a scalar):

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \frac{T}{2\pi} x_{n+1})$$
$$\dot{x}_{n+1} = \frac{2\pi}{T}$$

Suppose that the initial condition for the original nonautonomous system is $\mathbf{x}(t_0) = \mathbf{x}_0$. Then the initial condition for the reduced autonomous system can be chosen as: at new $t = 0$, $\mathbf{x}(0) = \mathbf{x}_0$ and $x_{n+1}(0) = (2\pi/T)t_0$.

Since $\mathbf{f}$ is cyclic in terms of t with the period T, $\mathbf{f}$ is cyclic in terms of x_{n+1} with the period 2π. Hence, x_{n+1} can be represented in the interval of $0 \le x_{n+1} < 2\pi$ by $x_{n+1} = 2\pi t/T \bmod 2\pi$. Using this transformation, results for autonomous systems can be applied to the time-periodic nonautonomous systems.

We note that a nonautonomous system which is not time-periodic can also be reduced to an autonomous system by $x_{n+1} = (2\pi/T)t$. However, the solution is necessarily unbounded, that is $x_{n+1} \to \infty$ as $t \to \infty$, hence many theoretical results concerning the steady-state behavior of autonomous systems do not apply. (For more, see Parker, 1989.) In the following, we will primarily discuss autonomous systems, unless otherwise stated.

Example 3. A forced damped pendulum

The equation of motion for a forced damped pendulum (Fig. 7.3) can be expressed in the following form.

$$\ddot{x} + a\dot{x} + b \sin x = c \sin(\frac{2\pi}{T} t)$$

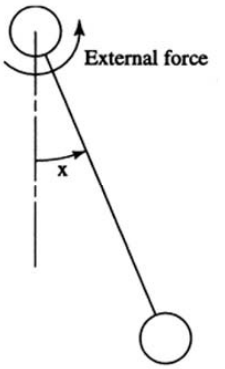


Fig. 7.3. A forced damped pendulum

where x is the only dependent variable representing the angle, and a, b, c, and T are parameters. (x, rather than θ, is used to be consistent with the above general notation.) In this equation, the first term represents acceleration (inertia), the second term the damping effect due to friction at the pivot, the third term gravity, and the right-hand side term an external force of sinusoidal torque applied at the pivot. This equation is second order since the term $\ddot{x}$, and nonautonomous since the right-hand side expression explicitly contains time t. We will reduce this equation to a set of first-order autonomous equations in two steps as follows:

Step 1. Second-order to first-order

Introduce a new variable y defined as $y = \dot{x}$ .Then the original equation becomes a set of two first-order equations:

$$\dot{x} = y$$

$$\dot{y} = -ay - b \sin x + c \sin(\frac{2\pi}{T} t)$$

Step 2. Nonautonomous to autonomous

Define a third variable z as $z = (2\pi/T)t$. Then the above set of two equations reduces to:

$$\dot{x} = y$$

$$\dot{y} = -ay - b \sin x + c \sin z$$

$$\dot{z} = 2\pi/T$$

Relationships between Discrete and Continuous Dynamical Systems

For certain cases, we can establish relationships between discrete and continuous

dynamical systems, although the corresponding counterparts represent different systems.

Discrete to continuous

Example 4. Compound interest

Let annual interest be i, and interest be compounded for every period h, which is a fraction of one year. For example, if the annual interest is 8%, i will be 0.08; if h = 0.25, it means that interest is compounded every quarter or three months. Let x_p be a deposit amount at time period p and x_{p+1} be the amount after one period. x_{p+1} then is the sum of the principal x_p at the beginning of the period plus interest during this period, ihx_p. Hence, the discrete difference equation is:

$$x_{p+1} = (1 + ih)x_p$$

Given an initial condition, x_0 at $p = 0$, we have $x_p = (1 + ih)^p x_0$, the amount after p periods. If we want to show the solution in terms of the number of years t, where $t = ph$, we have $x_t = (1 + ih)^{t/h} x_0$.

We now determine a continuous counterpart of the above difference equation in form of a differential equation. As before, let interest be compounded for every period h, and x_t be the amount at time t (in terms of the number of years). For x_{t+h}, the amount after one period of time length h starting from x_t, we have:

$$x_{t+h} = (1 + ih)x_t$$

It follows: $(x_{t+h} - x_t)/h = ix_t$. We consider reducing the compounding period to a very small amount of time; that is, interest is compounded every day, second, microseconds, and so on, to eventually zero, which means interest is compounded continuously (although no bank practices such continuous compounding). This corresponds to taking the limit of $h \to 0$. According to calculus, $\lim_{h\to 0} (x_{t+h} - x_t)/h$ is the definition of $\dot{x}_t$. Hence, we have:

$$\dot{x}_t = ix_t$$

With understanding that x is a function of t, we can also write the above as:

$$\dot{x} = ix$$

This is a continuous version of the problem. The meaning of this differential equation is that the rate of increase of the deposit amount is equal to the interest rate times the deposit amount. As before, given an initial condition, x_0 at $t = 0$, we have $x_t = e^{it} x_0$, the amount after t years.

In general, given a recurrence equation, compute $x_{t+1} - x_t$, the increase of x for unit time. Roughly, this corresponds to $\dot{x}_t$, the increasing rate of x. For example, in the above example, $x_{t+1} - x_t = ix_t$ represents the increase of interest for one year, and this corresponds to $\dot{x}_t$. The underlying (solution) function x_t for continuous values of t must be differentiable to make this correspondence. More precisely, we must compute $\lim_{h\to 0} (x_{t+h} - x_t)/h$ as in the above example.

Continuous to discrete

Given a continuous system, there are various ways to derive a discrete counterpart. Note that generally the continuous and discrete counterparts represent different systems. (For more details, see, e.g., Ott, 1993, pp. 9-10, and Parker and Chua, 1989, Chapter 2.)

Method 1. Select discrete time intervals, $t_k = t_0 + kT$, $k = 0, 1, 2, ...$ Using the given continuous system of equations and an initial condition at $t = t_0$, get a solution for $t = t_1$. Using the given continuous system of equations and the solution just obtained as a new initial condition at $t = t_1$, get a solution for $t = t_2$, and so on.

Method 2. Poincaré maps. Essentially, consider an (n - 1)-dimensional hyper-surface in the original n-dimensional state space. Pick discrete consecutive intersections of the solution (orbit) of the original continuous equations and the *hyper-surface*. Recall that the term "hyper" is used for a space dimension that is more than normal. For example, a hyper-plane or hyper-surface is an imaginary plane or surface whose actual dimension is three or more.

7.3 State and Phase Spaces

The variables x_t, y_t, or x, y, and so on, are called the **state variables** since they describe the states of the dynamical systems. A space made up of the state variables is called a **state space**. For example, for a two-dimensional system with the state variables x and y, a state space (or a state plane, particularly for a two-dimensional case) may have x for its abscissa, and y for its ordinate. Or, the two-dimensional space does not necessarily need to be represented by the Cartesian coordinate; it can be represented by, for example, a polar coordinate. Given a specific problem, one coordinate system is often more convenient than other systems.

A space made up of the state variables and their derivatives is called a **phase space**. In common practice the terms state and phase spaces are often used interchangeably. We will also follow this convention.

We can extend these spaces to higher dimensions. Since we live in three-dimensional space, when we go beyond the third dimension, we have a harder time drawing, visualizing, or understanding space. We often use our imagination to extend concepts in lower dimensions, such as line, surface, etc., to higher dimensions. Another way is to accept such a space as an abstract, formal extension of lower-dimensional spaces. Sometimes lower dimensional state and phase spaces can also be considered as abstract mathematical representations without graphical interpretation. In this section, we will study the behavior of dynamical systems in state and phase spaces.

7.3.1 Trajectory, Orbit and Flow

In a state (or phase) space, a path followed by a dynamical system as time progresses

is called a **trajectory** or **orbit**. A point on a trajectory corresponding to a specific time is called a **state point** or **phase point**. Fig. 7.4 depicts an example of a trajectory. Mathematically, a trajectory represents the path of the *solution* of a dynamical system, starting from a specific initial condition, in the state space. Some authors use "orbit" for discrete and "trajectory" for continuous systems. A **flow** is the group of trajectories generated by all the initial conditions in the state space. A flow is analogous to the paths followed by a flowing fluid.

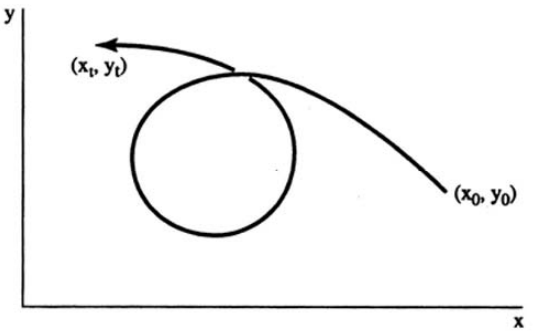


Fig. 7.4. An example of a state space and a trajectory

Example 5. A linearized loss-free, unforced pendulum

The equation of motion for this system is described by:

$$\ddot{x} + k^2 \sin x = 0$$

where k is a positive constant. This is a special case of the forced damped pendulum we have seen before (Fig. 7.3). In the above, the first term represents acceleration, and the second term gravity. Note that the two terms for damping due to friction and external force are dropped.

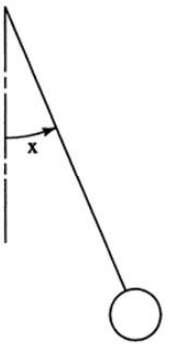


Fig. 7.5 A loss-free, unforced pendulum

When $|x|$ is small we can employ an approximation of $\sin x \approx x$, and the equation of motion becomes the following linear version.

$$\ddot{x} + k^2 x = 0$$

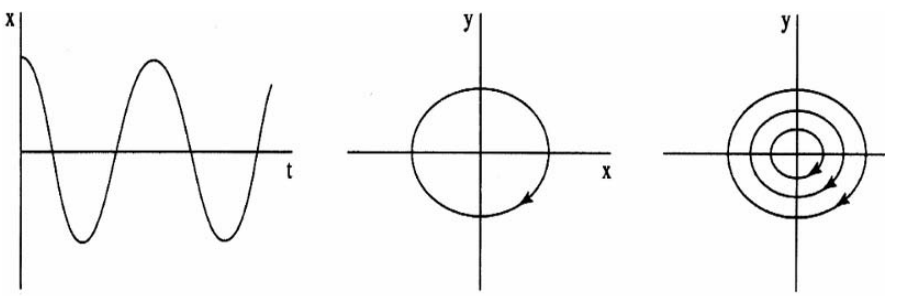
By introducing $y = \dot{x}$, this equation reduces to:

$$\dot{x} = y$$
$$\dot{y} = -k^2 x$$

The general solution is $x = a \cos(kt + b)$ and $y = -ak \sin(kt + b)$. An initial condition can be taken as holding, then releasing the pendulum at $x = x_0$ at the beginning, that is, $x = x_0$ and $\dot{x} = y = 0$ at $t = 0$. The particular solution corresponding to this initial condition is:

$$x = x_0 \cos(kt)$$
$$y = -kx_0 \sin(kt)$$

Fig. 7.6 (a) depicts $x = x_0 \cos(kt)$, that is, how x changes over time. From $x = x_0 \cos(kt)$ and $y = -kx_0 \sin(kt)$, we have $(x/x_0)^2 + (y/kx_0)^2 = 1$. That is, the solution or trajectory is an ellipsoid in the phase space as is shown in Fig. 7.6 (b). Note that in Fig 7.6 (b), time is implicitly included; for example, the point $x = x_0$ and $y = 0$ corresponds to $t = 0$, then periodic afterwards as, $t = 2\pi/k, 4\pi/k, \ldots$. Finally, Fig. 7.6 (c) shows a flow in the phase space, representing a set of solutions that corresponds to different initial conditions for x_0.



(a) (b) (c)

Fig. 7.6 Solution of a linearized loss-free, unforced pendulum. (a) A time waveform for x. (b) A phase space of x and y = $\dot{x}$ and a trajectory. (c) A flow

The basic concepts for representing dynamical systems by differential equations, their solutions and trajectories in the phase spaces, and flows discussed in the above example are the same for other problems. Determining solutions and drawing

trajectories, however, for certain problems can be much harder. In the following, we often skip some details of solutions for brevity.

In general, there are advantages and disadvantages for the two types of representations - time waveform (or series) and phase space. In the time waveform representation, since time is explicit, the behavior of state variables as time changes is clear. But, the relationships among the state variables and their derivatives are not obvious. The advantage and disadvantage for the phase space representation are just the opposite of the time waveform. In a phase space, time is not a part of the coordinates and it is included only implicitly in the phase space. Hence, a static diagram of a trajectory, such as Fig. 7.6 (b), does not fully illustrate the dynamic nature of the system. To understand the dynamic nature, we need a good imagination of a video where a phase point moves along the trajectory as time passes. On the other hand, the phase space representation often reveals characteristics which are not transparent from the time waveform.

7.3.2 Cobwebs

Cobwebs can be drawn for discrete dynamical systems, and they are somewhat analogous to phase spaces for continuous systems. That is, time is implicit in cobwebs; the behavior of state variables, which may not be evident from time series, may be revealed through cobwebs. Cobwebs are different, however, from phase spaces since the ordinates of cobwebs do not represent derivatives. Drawing cobwebs is straightforward and can be described as follows:

An algorithm for drawing a cobweb

Given: a discrete dynamical system $x_{t+1} = f(x_t)$ and an initial condition $x_t = x_0$ at $t = 0$.

(Initial setup before cobweb drawing) (See Fig. 7.7 (a))
Prepare a two-dimensional graph in which: Select x_t for abscissa and x_{t+1} for ordinate. Draw a graph of $x_{t+1} = f(x_t)$ and a 45-degree line of $x_{t+1} = x_t$ in the plane. Mark the initial condition $x_t = x_0$ on the abscissa.

(Iteration) (See Fig. 7.7 (b))

Repeat the following three steps for $t = 0, 1, 2, ...$, for necessary time intervals:

Step 1. Starting from x_t on abscissa, draw a vertical line until it intersects with the graph of $x_{t+1} = f(x_t)$. The ordinate of the intersection represents the value of x_{t+1}.

Step 2. Starting from the intersection, draw a horizontal line until it intersects with the 45° line of $x_{t+1} = x_t$. The abscissa of the new intersection now represents the value of x_{t+1} (since $x_t = x_{t+1}$).

Step 3. Advance t by 1, and go back to Step 1.

For easy recognition, the cobweb in Fig. 7.7 (b) is shown as bold line segments; if they are thin, they would look more closely to a real cobweb.

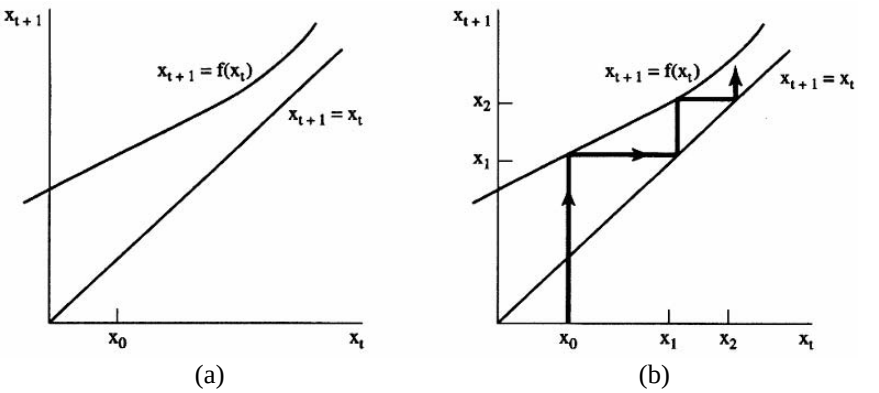


Fig. 7.7 Drawing a cobweb. (a) Initial setup. (b) Iteration steps for a cobweb, represented by bold line segments.

7.4 Equilibrium Solutions and Stability

If the value of a solution of a dynamical system, either for discrete or continuous, stays as a constant, it is called an **equilibrium solution** or a **fixed point**. The term a "fixed point" is often used in graphical context, such as an equilibrium solution in a state/phase space or a cobweb. However, these two terms, equilibrium solution and fixed point, are often used interchangeably and we will follow this convention.

There are two kinds of equilibrium solutions: stable and unstable. An equilibrium solution is **stable** if any solution in the near neighborhood of the equilibrium solution gets closer and closer to the equilibrium solution as time advances. A stable equilibrium solution or fixed point is also called an **attractor**, particularly in graphical context. This is because the flow is toward the fixed point. An equilibrium solution is **unstable** if any solution in the near neighborhood of the equilibrium solution tends to get far, at least a certain distance, from the equilibrium solution as time advances. An unstable fixed point is also called a **repeller**. For graphical representations, we will use solid black dots for stable equilibrium solutions, while open circles for unstable equilibrium solutions.

Discrete dynamical systems

Given a one-dimensional discrete dynamical system described by $x_{t+1} = f(x_t)$, suppose that a specific value of x, say, x_q satisfies the following:

$$f(x_q) = x_q$$

Then x_q is called an **equilibrium solution** of the discrete dynamical system.

The meaning of the equilibrium solution is as follows. Suppose that we start with an initial x_0 as x_q. Then $x_1 = f(x_0) = f(x_q) = x_q$, $x_2 = f(x_1) = f(x_q) = x_q$, and so on, hence, $x_t = x_q$ for any t. In words, the solution x_q stays the same as time changes - this is an equilibrium solution.

Extensions to higher-dimensional dynamical systems described generally by $\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t)$ work in the same fashion. That is, if a particular solution $\mathbf{x}_q$ satisfies $\mathbf{f}(\mathbf{x}_q) = \mathbf{x}_q$, then $\mathbf{x}_q$ is an equilibrium solution.

Example 6. Equilibrium solutions of a one-dimensional, linear discrete dynamical system

Consider a dynamical system,

$$x_{t+1} = f(x_t) = ax_t + b$$

where $a \neq 0$ and b are constants. An equilibrium solution can be found by solving

$$x_q = ax_q + b$$

This yields $x_q = b/(1 - a)$, assuming that $a \neq 1$. (If $a = 1$, we have a trivial case where the original equation becomes $x_{t+1} = x_t + b$. An equilibrium solution satisfies $x_q = x_q + b$, which implies $b = 0$ and x_q can be any value.)

To determine whether the equilibrium solution $x_q = b/(1 - a)$ is stable or unstable, we can solve the original recurrence equation as follows. By applying the recurrence equation for t, $t - 1$, $t - 2$, and so on, repeatedly, we have:

$$\begin{aligned}
x_t &= ax_{t-1} + b \\
&= a(ax_{t-2} + b) + b \\
&= a^2x_{t-2} + ab + b \\
&= a^2(ax_{t-3} + b)\, ab + b \\
&= a^3x_{t-3} + a^2b + ab + b \\
&= \ldots \\
&\quad \ldots \\
&= a^tx_{t-t} + a^{t-1}b + a^{t-2}b + \ldots + ab + b \\
&= a^tx_0 + b(a^{t-1} + a^{t-2} + \ldots + a + 1)
\end{aligned}$$

(By geometric progression formula we have:)

$$= a^tx_0 + \frac{b\left(1 - a^t\right)}{b\left(1 - a\right)}$$

Incidentally, this solution can also be proved to be correct by using mathematical induction. Using this result, we have

$$|x_t - x_q| = \left| a^t x_0 + \frac{b\left(1-a^t\right)}{b\left(1-a\right)} - \frac{b}{\left(1-a\right)} \right| = \left| a^t x_0 - \frac{ba^t}{\left(1-a\right)} \right|$$

$$= |a^t(x_0 - x_q)| = |a|^t \, |x_0 - x_q|$$

(The last equality holds since generally $|pq| = |p||q|$.) The behavior of $|x_t - x_q|$ depends on the value of a, and can be classified into the following three cases.

1) $|a| < 1$. *Stable*, since $\lim_{t\to\infty} |a|^t = 0$, and consequently $\lim_{t\to\infty} |x_t - x_q| = 0$.
2) $|a| > 1$. *Unstable*, since $\lim_{t\to\infty} |a|^t \to \infty$, and consequently $\lim_{t\to\infty} |x_t - x_q| \to \infty$.
3) $a = -1$. Then $x_1 = -x_0 + b$, $x_2 = -x_1 + b = -(-x_0 + b) + b = x_0$. Hence, $x_1 = x_3 = x_5 = \ldots = -x_0 + b$ and $x_2 = x_4 = \ldots = x_0$. An *oscillating solution.*

The following Fig. 7.8 shows a cobweb for a special case where, $a = -0.5$, $b = 3$, and $x_0 = 10$. The fixed point is $x_q = 2$ and it is stable.

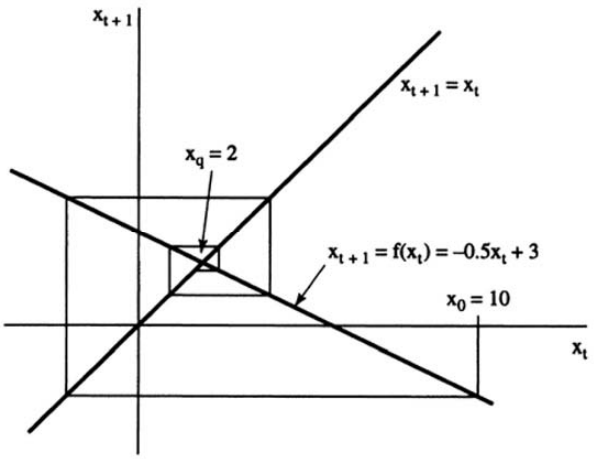


Fig. 7.8.A cobweb for a special case of a one-dimensional, linear discrete dynamical system, where $x_{t+1} = -0.5x_t + 3$ and $x_0 = 10$. The fixed point is $x_q = 2$ and it is stable.

Note that we have exactly one equilibrium solution in the above Example 6, since the recurrence equation $x_{t+1} = ax_t + b$ is linear, and consequently an equation for an equilibrium solution, $x_q = ax_q + b$, is also linear. Generally, a nonlinear dynamical system will have multiple equilibrium solutions. For example, a quadratic system, $x_{t+1} = ax_t^2 + bx_t + c$, will have two equilibrium solutions (unless the solutions are degenerated or complex numbers). A cubic system will have three equilibrium solutions, and so on. The more nonlinear the system is, the more equilibrium solutions and the harder to determine them.

Example 7. Loan payment

Let the loan balance at month $t = x_t$, monthly interest rate = r, and monthly payment = p. Then the balance at next month $t + 1$ will be the balance at t plus the interest for the month minus the payment, that is:

$$x_{t+1} = (1 + r)x_t - p$$

where $r, p > 0$. This is a special case of previous Example 6, in which $a = 1 + r > 1$ and $b = -p$. As before, an equilibrium solution can be found by solving $x_q = (1 + r)x_q - p$, which yields $x_q = p/r$. The meaning of this equilibrium solution is that if one makes a monthly payment of $p = rx_q$, which is exactly the same amount as interest, then the principle stays the same.

In Example 6, we found that when $|a| < 1$, the equilibrium solution is unstable. This can be easily understood by considering the behavior of x_t in the neighborhood of $x_q = p/r$. When $x_t < x_q = p/r$, that is $p > rx_t$, the payment is greater than interest, so the balance x_t will become smaller and smaller over time. This is a usual situation to return a loan. Eventually x_t will become zero, which means the loan is paid off, then payment will be stopped. For mathematical consideration, if payment is not stopped, $x_t \to -\infty$ or $|x_t - x_q| \to \infty$ as $t \to \infty$. When $x_t > x_q = p/r$, that is $p < rx_t$, the payment is smaller than interest, so the balance x_t will become greater and greater over time. So, $x_t \to \infty$ or $|x_t - x_q| \to \infty$ as $t \to \infty$. In both cases, either $x_t < x_q$, or $x_t > x_q$, a solution tends to get far from the equilibrium solution. Hence, the equilibrium solution is unstable and it is a repeller. Fig. 7.9 is the time series of the problem. Fig. 7.10 depicts cobwebs of the problem, one with an initial condition $x_{01} < x_q$, and the other with an initial condition $x_{02} > x_q$.

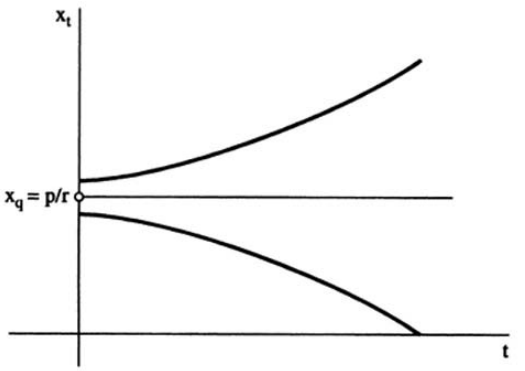


Fig. 7.9.Time series of loan payment. The fixed point $x_q = p/r$ is unstable; a solution tends to get far from x_q.

Continuous dynamical systems

Given a one-dimensional, continuous autonomous dynamical system described by $\dot{x} = f(x)$, suppose that a specific value of x, say, x_q satisfies the following:

$$\dot{x} = f(x_q) = 0$$

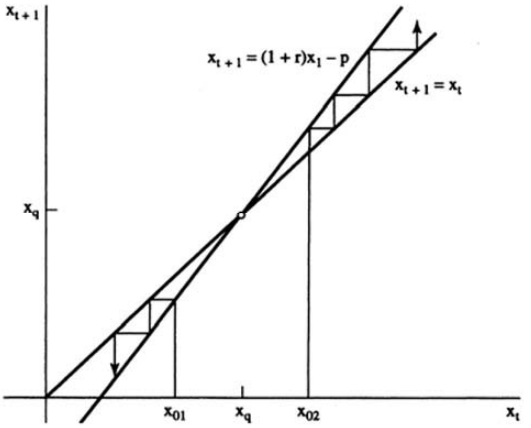


Fig. 7.10 Cobwebs of loan payment. One cobweb with an initial condition $x_{01} < x_q$, and the other with an initial condition $x_{02} > x_q$. x_q is an unstable fixed point.

Then x_q is called an **equilibrium solution** or a **fixed point** of the continuous dynamical system. The meaning of the equilibrium solution is that when we start with an initial x_0 as x_q, x stays as x_q for all subsequent t, since the time derivative of x with respect t is zero, that is, no change for x.

Extensions to higher-dimensional dynamical systems described generally by $= \mathbf{f}(\mathbf{x})$ work in the same fashion. That is, if a particular solution $\mathbf{x}_q$ satisfies $\mathbf{f}(\mathbf{x}_q) = 0$, then $\mathbf{x}_q$ is an equilibrium solution or a fixed point.

Example 8. Equilibrium solutions of a one-dimensional continuous dynamical system

Consider a dynamical system

$$\dot{x} = f(x) = (x^2 - 1)(x - 2)$$

Fig. 7.11 shows a phase space of the system. There are three fixed points in this system, corresponding to the three solutions for $f(x) = (x^2 - 1)(x - 2) = 0$, namely, $x = -1$, 1, and 2. When, for example, we start with x_0 equal to exactly -1, then x_t will stay -1 forever.

To determine the stabilities of these fixed points, let us interpret x as a position of a point on the abscissa, and $\dot{x}$ is the velocity of the point. When $\dot{x}$ is positive, i.e., the graph is above the abscissa, the point will move in the positive direction of x, or to the right on the x-axis. Similarly, when $\dot{x}$ is negative, i.e., the graph is below the abscissa, the point will move in the negative direction of x, or to the left on the x-axis. In addition, the higher the value of $|\dot{x}|$, the faster the velocity. The arrows on the abscissa in Fig. 7.11 show the directions and the magnitudes of the velocities at various values of x. For example, a point placed at slightly smaller than $x = 2$, say, x

= 1.99, will start moving slowly to the left, accelerate further to reach its highest (negative) velocity at $x = 1.5$, keep moving to the left, and eventually arrive at $x = 1$. A point placed at slightly larger than $x = 2$, say, $x = 2.01$, will start moving slowly to the right, accelerate further and further to fly away to infinity. Hence, the fixed point $x = 2$ is unstable. Similarly, we see that $x = -1$ is unstable, while $x = 1$ is stable.

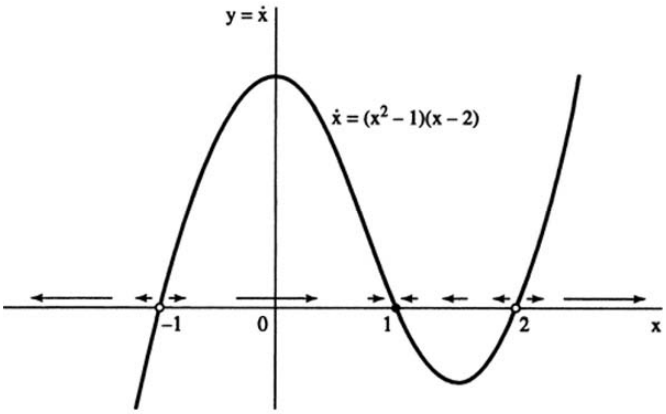


Fig. 7.11.A phase space illustrating fixed points and their stabilities of a dynamical system $\dot{X} = f(x) = (x^2 - 1)(x - 2)$. Arrows on the abscissa show the directions and magnitudes of velocities, $\dot{X}$, at various points of x.

7.5 Attractors

In the previous section, we learned that a stable equilibrium solution or a stable fixed point is called an attractor. The examples we have seen in the previous section are the simplest type of attractors, called **fixed-point attractors**. In general, an **attractor** is a set of points or a region in the state space where solutions that start in its neighborhood eventually converge to that set as time approaches infinity. Every attractor must not include another attractor as a subset, that is, every attractor must be minimal. In case of a fixed-point attractor, the set is a single point. There are other types of attractors, where an attractor is not a point; for example, it can be a loop or a set of confined endless spirals.

In this section, we study four types of attractors: 1) *fixed point*, 2) *periodic*, 3) *quasi-periodic*, and 4) *chaotic*, in increasing order of complexity. In most of the graphical representations, we focus the qualitative characteristics of the systembehaviors rather than quantitative, since the former is what we care about in many cases. Also in many cases resulting solutions are given without derivations since the process either require extensive analytical calculations or rely on numeric computations.

7.5.1. Fixed-point attractors

In the previous section, we have seen examples of fixed-point attractors. In these examples, solutions that start in the neighborhood of a fixed point have the fixed point as a steady-state solution. We will see another example of fixed-point attractors and introduce a few more related concepts.

Example 9. A damped pendulum

This is a special case of Example 3 for a forced damped pendulum, where the right-hand side term for external force is dropped. The equation of motion can be described as:

$$\ddot{x} + a\dot{x} + k^2 \sin x = 0$$

Fig. 7.12 depicts a phase space of this system.

As before, x represents the angle of the pendulum at the pivot and $y = \dot{x}$ the angular velocity. The origin is a fixed-point attractor; any solutions that start in the neighborhood of this fixed point will eventually converge to this point. Physically, this equilibrium solution represents that the pendulum comes to rest at its vertical position, that is, $x = 0$ and $\dot{x} = 0$, due to loss of energy from the friction. Generally, the set of points in phase space that are attracted to the same attractor is called the **basin of attraction**. The shaded region in Fig. 7.12 represents the basin of attraction to the origin. Similarly, there are other basins of attraction corresponding to other fixed points.

The point at $x = \pi$ and $\dot{x} = 0$, denoted by $\otimes$, is called a **saddle point**. It is a combination of an attractor and a repeller. The trajectories that are going towards the point are stable, while those going away are unstable. Imagine a horse saddle.

7.5.2. Periodic attractors

For a discrete dynamical system, $\mathbf{x}_{t+1} = \mathbf{f}(\mathbf{x}_t)$, if a solution $\mathbf{x}_t = \boldsymbol{\varphi}(t)$ satisfies:

$$\boldsymbol{\varphi}(t + T) = \boldsymbol{\varphi}(t)$$

for some fixed integer $T > 0$, then the solution $\boldsymbol{\varphi}(t)$ is called **periodic**. The smallest such T is called the **period** of the solution. Similarly, for a continuous dynamical system, $\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$, if a solution $\mathbf{x} = \boldsymbol{\varphi}(t)$ satisfies $\boldsymbol{\varphi}(t + T) = \boldsymbol{\varphi}(t)$ for some constant $T > 0$, then $\boldsymbol{\varphi}(t)$ is called periodic. The smallest such T is called the period of the solution.

For either discrete or continuous system, a periodic solution is **isolated** if there is no other periodic solution in the neighborhood. An isolated periodic solution is also called a **limit cycle**.

Certain dynamical systems have periodic rather than fixed point solutions as their steady-state solutions. In a phase space, this type of solution forms some form of a loop, such as a circle, ellipsoid, or deformed ellipsoid, etc., instead of a point. All

solutions that start at a point on the loop will move around on this loop. All solutions that start in the neighborhood of such a loop will eventually wind up on this loop and eventually move around the loop. This type of a steady-state solution is called a **periodic attractor**.

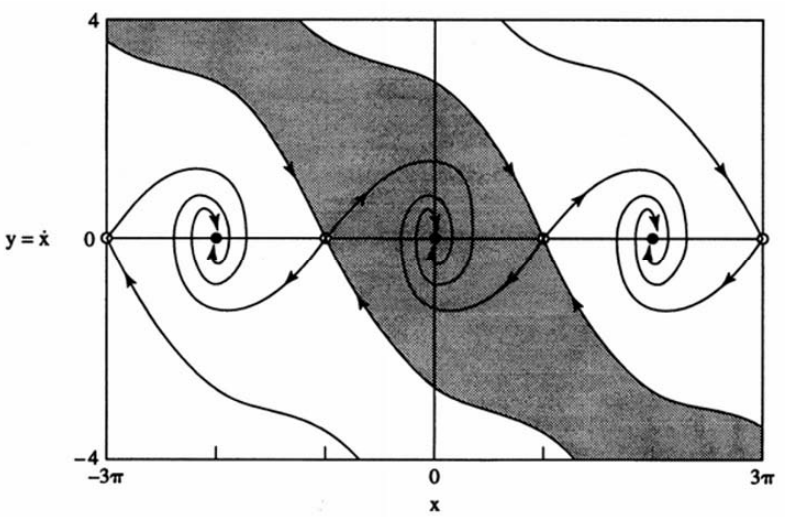


Fig. 7.12 A phase space for a damped pendulum system with $a = 0.4$ and $k = 1$.

Example 10. The van der Pol equation

A popular example of this periodic attractor is the following *van der Pol equation* of a continuous oscillatory dynamical system.

$$\ddot{x} + \varepsilon(x^2 - 1)\dot{x} + x = 0$$

where ε is a positive constant. The limit cycle depends on the value of the positive parameter ε. For example, when ε is small (e.g., $\varepsilon = 0.1$), the limit cycle is near a circle.

Fig. 7.13 shows a phase space for $\varepsilon = 1$. The heavy curve represents the limit cycle. Neighborhood solutions represented by thin curves approach the limit cycle as time advances. The second term, $\varepsilon(x^2 - 1)$, is a nonlinear damping term. When $|x| > 1$, this term is positive and acts as ordinary damping, causing a decay of the motion. When $|x| < 1$, the term becomes negative, and it acts as "negative damping" or positive pumping, exciting the motion. Thus, a solution that starts at a large value of x will be contracted to the limit cycle. Similarly, a solution that starts at a small value (excluding the origin, which is an unstable equilibrium solution) of x will be expanded toward the limit cycle. The system eventually settles to the steady state limit cycle, where the ordinary and negative damping effects balance over one period.

7.5.3. Quasi-periodic attractors

Steady state solutions for some dynamical systems are neither fixed points nor periodic as discussed above, but referred to as **quasi-periodic**. The name quasi-periodic indicates a solution that may appear periodic at glance, but the trajectory never repeats itself.

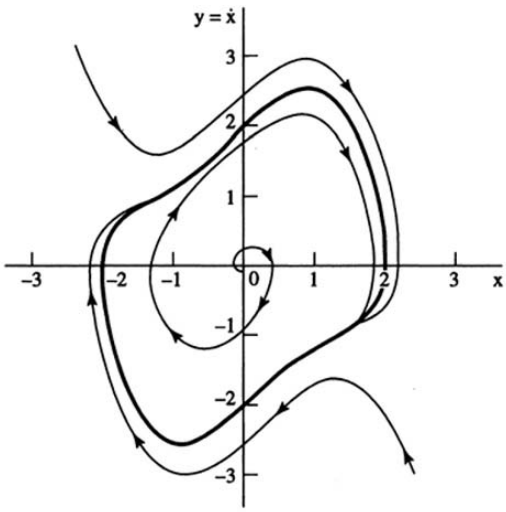


Fig. 7.13. Phase space for the van der Pol equation, with $\varepsilon = 1$. The heavy curve shows the limit cycle.

To illustrate quasi-periodic solutions, we introduce a new two-dimensional state space representation called **torus** (Fig. 7.14). We have been using the Cartesian coordinates for state spaces - for example, for two-dimensional space, the abscissa for x and the ordinate for y. In the torus, the two dependent variables on time t are angles denoted as θ_1 and θ_2. We could use x and y instead of θ_1 and θ_2 as state variables, but since they are angles rather than distance, the latter would be more descriptive. A point in this space is a point on the surface of the torus determined by values of θ_1 and θ_2. θ_1 is a vertical angle measurement analogous to longitude and θ_2 is a horizontal angle measurement analogous to latitude. Using the same analogy, there are outer and inner equators on the torus. The circle on the top of the torus would be the "north pole." Note that the north pole on the torus is not a point but a circle. Similarly, the bottom circle would be the "south pole."

We may arbitrarily choose the front point on the outer equator (i.e., the closest point to us) as the origin, $\theta_1 = 0$ and $\theta_2 = 0$. Then along the outer equator, the right, back, and left points correspond to $\theta_2 = \pi/2$, π, and $(3/2)\pi$, respectively. At any vertical cross section, $\theta_1 = 0$, $\pi/2$, π, and $(3/2)\pi$ correspond to the outer equator, north pole, inner equator, and south pole, respectively.

When a solution of a dynamical system, $\theta_1(t)$ and $\theta_2(t)$, and an initial condition, for

example $\theta_1 = 0$ and $\theta_2 = 0$ at $t = 0$, are given, we can draw the trajectory on the surface of the torus. We can imagine a point starts at the origin, $\theta_1 = 0$ and $\theta_2 = 0$, then moves along the trajectory as time advances.

Case study. A simple dynamical system in the torus

To understand quasi-periodic, we will study the following simple example.

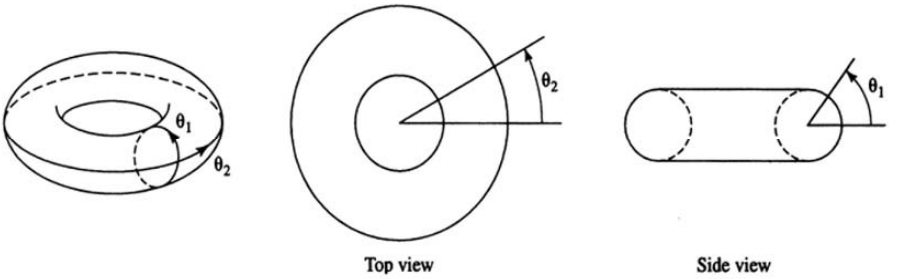


Fig. 7.14. The torus, a two-dimensional state space representation.

$$\dot{\theta}_1 = \omega_1$$
$$\dot{\theta}_2 = \omega_2$$

where ω_1 and ω_2 are positive constants. Since $\dot{\theta}_1$ and $\dot{\theta}_2$ represent angular velocities, a point in the system moves with constant angular velocities of ω_1 and ω_2. Note that to determine a trajectory of the system, only the ratio, $r = \omega_1/\omega_2$, is needed, rather than specific values of ω_1 and ω_2. Specific values of ω_1 and ω_2 will affect only how fast the point moves on the trajectory, not the shape of the trajectory. Geometrically, the ratio, $r = \omega_1/\omega_2$, represents the "angular slope" of the trajectory. In the following, we will see three cases in terms of r: 1) $r = 1$, 2) r is a rational number, and 3) r is an irrational number. The first and second cases are essentially the same, but we start with the simplest case 1 for easy understanding. The last case is the one for quasi-periodic.

1. $r = \omega_1/\omega_2 = 1$. In this case, the solution is a single closed loop. For example, let us assume $\omega_1 = \omega_2 = (\pi/2)$/sec, that is, both θ_1 and θ_2 advance $\pi/2$ radian = 90° for every second. As mentioned above, although specific values of ω_1 and ω_2 are not needed to determine the trajectory, they will make us easy to understand how θ_1 and θ_2 change at specific time. Imagine a point starts at the origin at $t = 0$. At the beginning, the point will move continuously to the upward right on the torus. At $t =$ 1 second, the point will be at $\theta_1 = \pi/2$ and $\theta_2 = \pi/2$, the right side of the north pole circle. At $t = 2$ seconds, the point will be at $\theta_1 = \pi$ and $\theta_2 = \pi$, the inner equator on the back. At $t = 3$ seconds, the point will be at $\theta_1 = 3\pi/2$ and $\theta_2 = 3\pi/2$, the left side of the south pole circle. At $t = 4$ seconds, the point will be back to the origin where it was before at $t = 0$ second. The following is a summary of the movement of the point at these selected time in the above.

Time t	0	1	2	3	4
θ_1	0	$\pi/2$	π	$3\pi/2$	0
θ_2	0	$\pi/2$	π	$3\pi/2$	0
Descriptive Location	front, center	right north pole	back, inner equator	left south pole	front center

2. $r = \omega_1/\omega_2$ is a rational number, i.e., r can be represented as $r = p/q$, where p and q are integers with no common factors. Let us pick out specific values for an illustration purpose: $p = 2$, $q = 3$; $\omega_1 = (2\pi/3)$/sec and $\omega_2 = \pi$/sec. The following table shows how θ_1 and θ_2 change over time. Their values are computed by taking modulus 2π, except that when the value is 2π, it is left as 2π.

Time t	0	1	2	3	4	5	6
θ_1	0	$2\pi/3$	$4\pi/3$	2π	$2\pi/3$	$4\pi/3$	2π
θ_2	0	π	2π	π	2π	π	2π

We observe that one period is 6 seconds; during one period, θ_1 completes two revolutions and θ_2 completes three revolutions. Fig. 7.15 is a rough sketch of the trajectory. Details of the trajectory are not an important issue here. Rather, the key is that the trajectory is a closed loop, that is, the solution is periodic.

We now extend this observation to general case of $r = p/q$. When θ_1 completes p revolutions and θ_2 completes q revolutions, they are back to the same starting point for the first time. This is because p revolutions for θ_1 and q revolutions for θ_2 require exactly the same amount of time; no smaller numbers of revolutions of θ_1 and θ_2 will result in the same time since p and q do not have a common factor. Again, the trajectory is a closed loop.

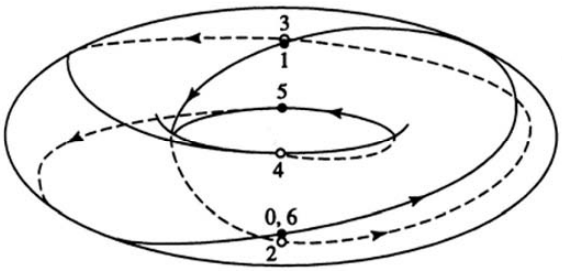


Fig. 7.15. A trajectory for $r = \omega_1/\omega_2 = 2/3$.

3. $r = \omega_1/\omega_2$ is an irrational number, i.e., r *cannot* be represented as $r = p/q$, where p and q are integers. Examples of irrational numbers are $\sqrt{2}$ = 1.4142..., and π = 3.14159.... From the above case 2, if $r = p/q$, then after θ_1 completes p revolutions and θ_2 completes q revolutions, they are back to the same starting point. If r cannot

be represented in form of p/q, there are no integer revolutions for θ_1 and θ_2 complete at the same time. This implies that the moving point will never be back where it started. Furthermore, by shifting any other location on the trajectory as an initial point, we conclude that no part of the trajectory intersects to itself. That is, the trajectory never repeats. This means that the moving point winds around forever endlessly. This solution is called quasi-periodic.

7.5.4. Chaotic attractors

A steady state solution that is bounded and is none of the previous three types of attractors - fixed-point, periodic, and quasi-periodic, is called a **chaotic attractor** or a **strange attractor**. A solution is **bounded** when it is confined in a finite region in the state space without diverging to infinity. The exact definition of these terms, chaotic and strange attractors, sometimes varies by authors and has changed over years. The basic concept is, however, that it exhibits an *aperiodic* and *highly irregular geometric pattern* and also is *sensitive to an initial condition*. **Aperiodic** is the term used when neither fixed-point, periodic, nor quasi-periodic apply.

At the beginning of this chapter we described the typical features of chaotic systems. There is no standard definition of chaos and it has been described in different ways. One way is to define **chaos** as a dynamical system whose steady state solution is a chaotic attractor. Sometimes the terms of "chaos" and "chaotic attractor" are used interchangeably. Chaotic attractor often refers to the geometric configuration in state space; chaos is used to describe the dynamics, that is, the motion of the solution as time progresses on a chaotic attractor.

As an example of a chaotic attractor, we will discuss the Lorenz attractor. In his seminal article (Lorenz, 1963), he described chaotic dynamics for a toy model of weather patterns. This was the first introduction of the concept of chaos in modern history, although the term chaos was not born at that time.

Example 11. Lorenz attractor

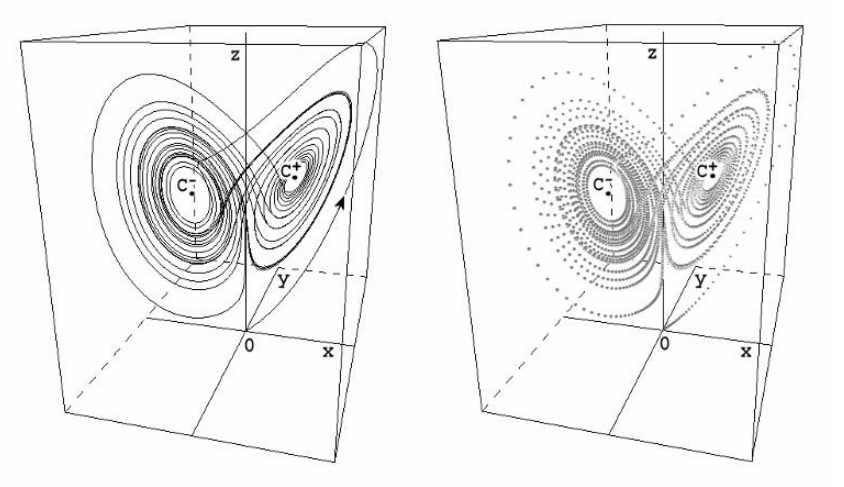
The dynamical system is described by the following Lorenz equations:

$$\dot{x} = \sigma(y - x)$$
$$\dot{y} = rx - y - xz$$
$$\dot{z} = xy - bz$$

Here $\sigma, r, b > 0$ are parameters related to the weather pattern model. We note that the system is three dimensional and a relatively simple nonlinear system where the only nonlinearity is for the quadratic terms xz and xy. We also note that x and y are symmetric with respect to the origin, since the system stays the same when we replace (x, y) with $(-x, -y)$.

Fig. 7. 16 (drawn by my graduate student Todd Posius) depicts the Lorenz attractor for $\sigma = 10$, $r = 28$ and $b = 8/3$. Fig. (a) is an actual solution while Fig. (b) plots only selected points with a fixed time interval. The attractor looks like a pair of butterfly wings. The points C^+ and C^- are unstable fixed-point attractors. In the figure we can see a typical behavior of a solution. The solution that starts near the origin rises to the right, then plunges to near C^-. After slow spiral outward, the state point

moves back to the right, spirals around, moves back to the left, and so on indefinitely. The motion is aperiodic and highly irregular for both its trajectory and speed. The trajectories never intersect themselves. Solutions are sensitive to the initial conditions; trajectories that start at slightly different points separate at exponential rate. Long term prediction is essentially impossible due to the sensitivity to initial conditions, which can only be known to a finite precision.



(a) (b)

Fig. 7.16. (a) The Lorenz attractor and a typical trajectory. (b) Only selected points in (a) are plotted with a fixed time interval.

7.6 Bifurcations

So far we have focused on the study of dynamical systems for fixed values of parameters. For example, for a dynamical system, $x_{t+1} = rx_t(1 - x_t)$, where r is a parameter, we would select a specific value for r, say, $r = 2$, then we can determine the solutions or time series for various initial values of x_t.

What will happen if we change the value of the parameter, say, from $r = 1.5$ to 2, 2.5, 3, 3.5, and so on? This may add a new "dimension" to the problem. That is, we can vary the value for the parameter, select different values for the initial condition, and investigate the behaviors of the solutions as time progresses. In this way, we may be studying an enormous number of cases since there are so many combinations for different values of the parameter, initial conditions, and time. When we focus on only steady-state solutions, the number of combinations will be much less since the time factor will be eliminated. Furthermore, if initial conditions do not affect the steady-state solutions, the problem will be much simpler. In such a case, we can focus our attention on the relationship between the steady-state solution versus the parameter. Whether initial conditions affect the steady-state solution depends on the

dynamical systems.

Generally, when we change a parameter value, say, from $r = 2.5$ to 2.6, the value of the fixed-point steady-state solution, x_∞ or simply x, may change *quantitatively*, say, from $x = 0.5$ to 0.55. At some point of the parameter value, the solution may *qualitatively* change from fixed-point to periodic, or periodic to chaotic. Or, in certain cases, a fixed point may suddenly disappear or may be created.

A **bifurcation** refers to a sudden, qualitatively different appearance in the solution of a nonlinear dynamical system as a parameter is varied. The parameter value at which a bifurcation occurs is called a **bifurcation point**. In a commonly used term, a bifurcation means to divide into two branches. At a bifurcation in a dynamical system, a fixed-point steady-state solution may split into a period-2 cycle steady-state solution (which will be discussed soon). This means that the dynamical system with an equilibrium solution will no longer have such a unique steady-state solution after the parameter passes a certain value. Furthermore, a period-2 cycle solution may change to period-4, period-4 to period-8, and so on, and finally, a periodic solution may change to chaos at bifurcations.

In the following, we will study the bifurcations using a relatively simple equation called the logistic equation. This was published by Robert M. May (1976) which is one of the major seminal articles in the history of chaos research. As May points out, this is an example in which a simple equation can demonstrate complicated dynamics.

The logistic equation

The following one-dimensional discrete system is called the **logistic equation** or **logistic map**:

$$x_{t+1} = rx_t(1 - x_t)$$

where $r \geq 0$ is a parameter. This equation is quadratically nonlinear since $rx_t(1 - x_t) = rx_t - rx_t^2$. The equation embodies a simple model of population fluctuations in an ecological system. Here r represents the population growth rate and x_t represents the normalized population of a species of the t-th generation or year t. For example, if the maximum real population is one million, dividing by one million will yield a normalized value of the population between 0 and 1. The quadratic expression $x_t(1 - x_t)$ takes the maximum value of 1/4 at $x_t = 1/2$. Hence, for x_{t+1} to stay within its range, i.e., $x_{t+1} \leq 1$, r must be $r \leq 4$. (For example, if $r = 5$, then for $x_t = 1/2$, x_{t+1} will be 1.25 which exceeds the upper bound 1.) Hereafter, we restrict r to $0 \leq r \leq 4$.

In this model the population level of the $t + 1$ generation, x_{t+1}, is proportional to the previous population x_t. This is because the population of the current generation is likely to be proportional to the number of matings and consequently the number of resulting offspring. x_{t+1} is also assumed to be proportional to $(1 - x_t)$ since as one approaches population saturation, the population tends to decrease due to worsening environmental condition (such as lack of food). This also serves as a simple model for other systems such as products in the market place.

Fig. 7.17 is a sketch of the steady-state solution, x_∞, versus the parameter r. This figure turns out to be not as complicated as it would since it does not depend on the initial value x_0. (If it does, we would have to draw a three-dimensional figure for x_∞ on r and x_0.) Now let us go over on Fig. 7.17 for various values of r, starting from $r = 0$.

For convenience, we divide the interval of $0 \le r \le 4$ into the following four domains, I, II, III and IV:

Domain	Interval of r	Characteristic of Steady-State Solution, x_∞
I	$0 \le r < 1$	$x_\infty = 0$, that is, zero steady-state solutions represent the species' extinction because of the low growth rate.
II	$1 \le r < 3$	Nonzero fixed-point steady-state solutions. The higher the value of r, the higher the value of x_∞; that is, the higher the growth rate, the more the steady-state population. For $r = 2.9$ the population x_∞ saturates at a fixed-point steady-state value $x_\infty \approx 0.655$. We note that so far variations on r resulted in changes in the values of the solutions, but not on the essential characteristics of the solutions.
III	$3 \le r < 3.57$	As seen in Fig. 7.17, at $r = 3$ the steady-state solution splits into two values of x_∞, the first encounter of a bifurcation. For example, say, $r = 3.2$, the steady-state solution never reaches a fixed point; instead, x_∞ alternates between two values, e.g., $x_t = x_{21} \approx 0.513$, $x_{22} \approx 0.799$, $x_{23} \approx 0.513$, $x_{24} \approx 0.799$, and so on. This situation is easier to understand in the time series depicted in Fig. 7.18 (a). We recall that although only the discrete corner points of the time-series graph are meaningful, it is a common practice to connect these points with line segments for easier recognition of the sequences. This type of oscillatory solution, where x_t repeats every two iterations, is called a **period-2 cycle**. As seen in Fig. 7.17, within the range of r for period-2 cycles, the higher the value of r, the further apart the two values of x_∞ are. For example, when r is close to 3, both values of x_∞ are close to the single fixed-point value of x_∞ near $r = 3$. The difference between the two x_∞ values become larger when r increases. A graph like Fig. 7.17 is called a **bifurcation diagram**. When r increases more and reaches to $r = 3.449$, the steady-state solution splits into four values of x_∞. This is called a **period-4 cycle**, where x_t repeats every four iterations. A time series of this situation is shown in Fig. 7.18 (b). For example, for $r = 3.5$, x_∞ may repeat every four iterations as, $x_\infty = 0.383$, 0.827, 0.501, and 0.875. Further **period-doubling** to cycles of period 8, 16, 32, ..., occur as r increases as $r = 3.544$, 3.564, 3.569, and so on. Successive bifurcations come faster and faster.

Domain	Interval of r	Characteristic of Steady-State Solution, x_∞
IV	$3.57 \le r \le 4$	Finally at $r \approx 3.57$, the steady-state solution bursts into the aperiodic, that is, chaos. Here the steady-state solution x_∞ changes from t to $t + 1$ irregularly (Fig. 7.18 (c)); in this way, x_∞ takes infinitely many different values, and the whole regions of the bifurcation diagram are blacked in. An intuitive interpretation of the behavior of the steady-state solutions in terms of r may be given as follows. Mathematically, r represents the degree of nonlinearity of the system. The higher the value of r, the higher the nonlinearity, thus resulting in more and more complex solutions, that is, from fixed-point to periodic and further to chaos. The story, however, does not end here. When r is further increased to, for example, $r \approx 3.68$ or 3.83, in the middle of chaos, cyclic solutions suddenly return. Period-doubling bifurcations begin with odd periods such as 3, 6, 12, ..., or 7, 14, 28, ... Successive bifurcations come faster and faster as before, and then breaking off once again to renewed chaos. Although this fine structure is not shown in Fig. 7.17, a magnified diagram resembles the earlier bifurcations.

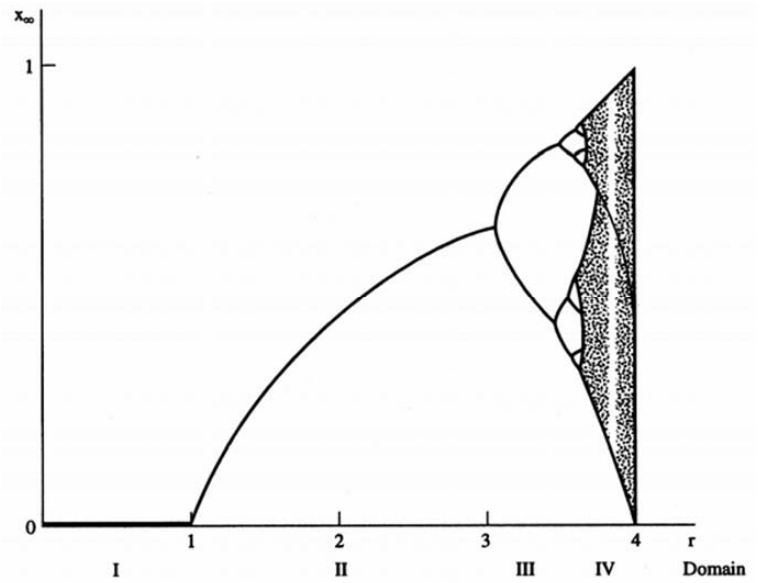


Fig. 7.17. A rough sketch for the steady-state solution x_∞ versus the parameter r for the logistic equation: $x_{t+1} = rx_t(1 - x_t)$.

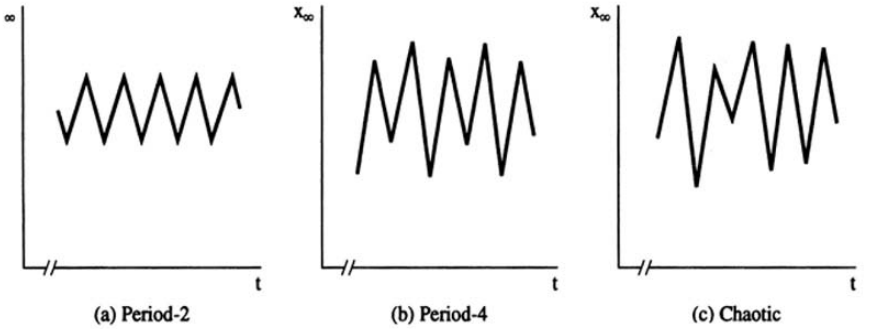


Fig. 7.18.Time series for period-2, period-4, and chaotic steady-state solutions: X_∞ versus t.

Incidentally, the fundamental idea of chaos control techniques is to apply a very small feedback perturbation to a system-wide parameter such as r or to a system variable such as x_t. This process makes the sensitivity to initial conditions or perturbations inherent in the chaos work for us. In comparison to linear systems where small controls yield small results, chaos enables small controls to have enormous effects.

7.7 Fractals

A **fractal** is a geometric object that has the following properties:

1. *Recursive self-similarity*. Any magnification of a part of the object has the same or similar shape as the original shape on arbitrarily small scale. As a result, a fractal is an extremely irregular curve or surface formed of an infinite number of similarly irregular sections.

2. *A non-integer, that is, fractional dimension*. Instead of an integer dimension such as 1, 2 or 3, a fractal may have a fractional dimension such as 1.26 or 2.05.

We may wonder why fractals are related to chaos. Fractals are geometric objects of static images, while chaos is a dynamical system in which the motion of a point is our interest. It turns out that fractals and chaos are closely related. Almost all chaotic (strange) attractors are fractals. Fractals also occur in bifurcation diagrams as studied at the end of the last section. In the following, we discuss fundamental basic ideas of these properties of fractals using a simple example, called the von Koch curve.

The von Koch curve

Construction of the von Koch curve is illustrated in Figs. 7.19 and 7.20. Fig. 7.19 shows the basic operation for constructing the curve. Given a line segment, we take the middle third as the base of an equilateral triangle and replace it with the other two sides of the triangle. This basic operation is repeated recursively for newly generated line segments to finally obtain the von Koch curve as is shown in Fig. 7.20.

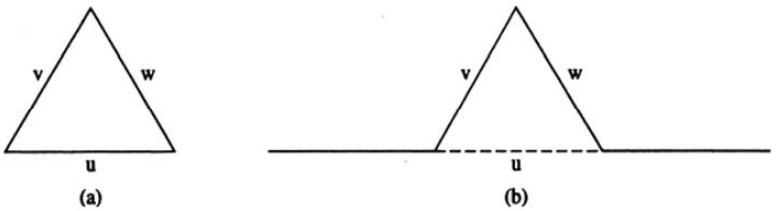


Fig. 7.19.The basic operation for constructing the von Koch curves. (a) An equilateral triangle with three sides *u* (base), *v* and *w*. (b) Consider the middle third of a line segment as side *u*, and replace it with the two other sides *v* and *w*.

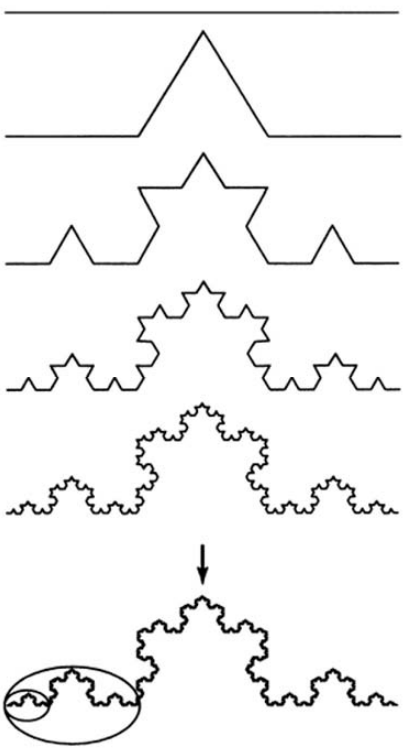


Fig. 7.20.Successive construction of the von Koch curve, starting from the top to the von Koch curve at the bottom.

In the von Koch curve shown in the bottom of Fig. 7.20, we see the part inside of the larger ellipse has the same shape as the original, that is, the part is a miniature of the original. Similarly, the part inside the smaller ellipse again has the same shape as the one inside the larger ellipse, which in turn the same shape as the original shape.

That is, this is recursive self-similarity for fractals. In the following, we will discuss fractional dimensions, the second property for fractals, using the von Koch curve as an example.

Fractal dimensions

We have been using the term "dimension" to represent a geometric extent. In Euclidean (Cartesian) space, the dimension is 1 when the space is a line, it is 2 for a plane, and it is 3 for a solid. Extending this idea slightly, a dimension may be said to be the minimum number of coordinates or variables needed to represent every point in space. For example, the dimension of a smooth curve is 1, since every point in the space (i.e., on the curve) can be represented by one number, the distance along the curve from its origin (Fig. 7.21).

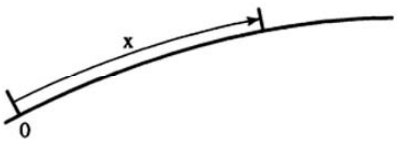


Fig. 7.21. The dimension of a smooth curve is 1.

More generally, a dimension can be some kind of measure that characterizes the space-filling properties for a set of points in space. Using this broader interpretation of dimensions, we can define other types of dimensions including fractional ones. The generic term that allows fractional values is **fractal dimension**. Different types of dimensions can be employed within fractal dimensions by using different definitions. Now we will discuss what dimension is appropriate for the von Koch curve. We claim that the dimension is neither 1 nor 2, but fractal. In Fig. 7.19 (b), we see that the total length of line segments is increased by 4/3 times by this operation. This is because the middle 1/3-long section is replaced by 2/3-long hat-shaped sections. Therefore, suppose that the length of the initial line in Fig. 7.20 is 1. Then the second shape has total length of 4/3, the third pattern has $(4/3)^2$, ..., and n-th pattern has $(4/3)^{n-1}$. Hence, the total length of the von Koch curve will approach $(4/3)^{\infty} = \infty$ when $n \to \infty$. As we saw before, the von Koch curve is recursively self-similar on an arbitrarily small scale. Hence, the length of every point on the curve is infinitely far from every other. This suggests that the dimension of the curve cannot be 1, since every point on the curve cannot be represented by one number. The dimension does not appear to be 2, since the curve does not have any area. To solve this problem, we introduce a new concept called similarity dimension, a simple fractal dimension.

Similarity dimension

Consider partitioning a d-dimensional shape, such as a line segment, square or cube in Euclidean space, into p similar shapes, scaled down by a factor of r. We see the relationship among d, p, and r is $p = r^d$ (Fig. 7.22). Although cases for $d = 3$ are not shown in Fig. 7.22, the same relationship holds for $d = 3$; for example, when $r = 2$, a cube is divided into $p = r^d = 2^3 = 8$ scaled-down cubes.

We extend dimension d that satisfies the relationship $p = r^d$ from integer to include fractal dimensions, called **similarity dimensions**. By taking the logarithms of $p = r^d$ with any base, we have: $\log p = d \log r$, or by arbitrarily choosing the natural logarithm, ln, we have $\ln p = d \ln r$. By explicitly writing in terms of d, we define similarity dimension as:

$$d = \frac{\ln p}{\ln r}$$

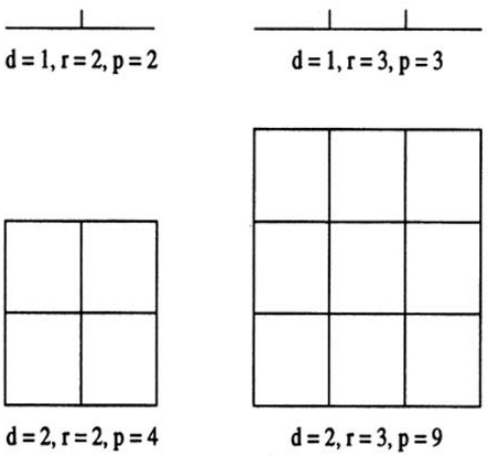


Fig. 7.22.Partitioning a d-dimensional shape into p similar shapes, scaled down by a factor of r. We see that $p = r^d$ holds.

In case of the von Koch curve, $r = 3$ and $p = 4$. Therefore, the similarity dimension of the von Koch curve is $(\ln 4)/(\ln 3) = 1.26186$. As mentioned earlier, other fractal dimensions can be defined, and dimensions of chaotic shapes, such as chaotic attractors and bifurcation diagrams, can be determined.

The Mandelbrot set, impressive fractal images, has been developed by using computer graphics around 1980 (see for example Gleick, 1987 for colorful images). This set together with the concept of fractal geometry were major historical developments in the study of chaotic systems.

7.8 Applications of Chaos

Practical applications of chaos are new and many potential ones are being investigated. In this section, we will briefly discuss the basic ideas of various types of chaos applications. For easy understanding, we classify the application types into the following four types.

1. Control and stabilization
2. Synthesis
3. Analysis and prediction
4. Hybrid systems

After the discussions of these four types of chaos applications, we will list sample potential application areas in various disciplines.

Control and stabilization

The extreme sensitivity of nonlinear or chaotic systems to tiny perturbations can be manipulated to stabilize or control the systems. The fundamental idea is that tiny perturbations can be artificially incorporated to either control or stabilize a large system - to direct a chaotic system into a desired state (control) or to keep it stable (stabilization). Typically such control or stabilization operations require much smaller amounts of energy than for nonchaotic systems. A good example of this type of application is the following NASA satellite control.

NASA satellite control. In 1978 a spacecraft called the International Sun-Earth Explorer-3 (ISEE-3) was launched toward a halo orbit around a Sun-Earth libration point. In 1983, ISEE-3 was retargetted for an interception with a comet million miles across the solar system. NASA engineers performed this orbital acrobatics by a combination of propulsive maneuvers, lunar swingbys, and the effective use of solar perturbations. Although the term "chaos control" did not exist at that time, this event was a demonstration of the idea. Through clever burns of fuel, the engineers exploited the chaotic sensitivity to perturbations exhibited by the three celestial body problem to use small amounts of fuel (all that was available) to nudge the spacecraft near the desired comet to be investigated. If the system were nonchaotic, large perturbations and subsequent large expenditures of fuel would have made such a mission impossible to achieve.

Other potential applications of small carefully chosen chaos control interventions include more efficient airplane wings, power delivery systems, turbines, chemical reactions in industrial plants, combustion, implantable heart defibrillators, brain pacemakers, conveyor belts, and economic planning.

Synthesis

Artificially generated chaotic outputs may be applied to certain types of problems to make the systems, either chaotic or non-chaotic, work better. The fundamental idea

is that regularity is not always the best, depending on the types of the problems. Artificially stimulated chaotic brain waves may some day help break up epileptic seizures by keeping the brain away from undesired periodicity. We can synthetically generate chaotic output for consumer products, such as air conditioners and fan heaters, to increase natural feeling for human comfort.

Two identical (called "synchronized") sequences of chaotic signals can be used for encryption by superposing a message on one sequence. Only a person with the other sequence can decode the message by subtracting the chaotic masking component. In communications, an artificially generated chaotic signals can follow a prescribed sequence, thus enabling to transmit information. Artificially generated chaotic fluctuations can be used to stimulate trapped solutions such that they escape from local minima for optimization problems or learning as in neural networks.

Analysis and prediction

When we can successfully model a chaotic system, it can be used to analyze the time series of the system. It then can lead to better understanding or more efficient design of the system. Or, it can be used for prediction or detection of system's behavior in the near future. For example, signals of a system, such as machine operations or human body, are carefully analyzed to detect a near future failure, thus avoiding a much more costly repair process. Other potential application domains include contagious disease incidence, cardiology, ecology, financial market, economy, fluid flow, weather, and climate such as the El Niño oscillation.

Hybrid systems

Other areas of AI, such as (artificial) neural networks, genetic algorithms, and fuzzy logic can be employed together with chaotic systems, as, e.g., neural networks + chaos, or neural networks + fuzzy + chaos. The fundamental concept of such hybrid systems is to complement each other's strengths, thus creating new approaches to solve problems.

Neural networks + Chaos Neural networks are modeled on biological neural networks or the human brain, and the brain exhibits chaotic behavior. Hence, it is natural to incorporate chaos to the study of neural networks. The chaotic biological system is capable of performing complex tasks such as speech production, visual and audio pattern recognition, and motor control, with flexibility. These tasks are hardly achieved by regular engineering systems. Artificial chaotic systems have been incorporated with the backpropagation model and associative memory. From engineering point of view, such chaotic neural network may be applied for prediction and control. From a scientific point of view, such network might lead to a better understanding of a biological neural network where the normal brain exhibits chaos. Also, as discussed before, artificially generated chaotic fluctuations can be used to escape from local minima for certain types of neural networks.

Genetic algorithms + Chaos Potential applications include the use of chaos as a tool to enhance genetic algorithms. For example, certain chaotic functions, rather than random numbers, might be used in the processes of crossover, etc. This may alter the

characteristics of genetic algorithm solutions, hopefully toward more desirable situations such as avoiding premature convergence. This may be interpreted as the use of artificially generated chaotic functions to escape from local minima.

Chaos modeling of genetic algorithms can be another example of the potential use of chaos as a tool to analyze genetic algorithms. Genetic algorithms, especially for those that generate chaotic solutions, may be analyzed by a chaos model. Conversely, genetic algorithms can be a useful tool to describe a complex chaotic system where common mathematical modeling is difficult.

Fuzzy logic + Chaos Fuzzy systems are suitable for uncertain or approximate reasoning, especially for systems for which a rigorous mathematical model is difficult to derive. They also allow us to represent descriptive or qualitative expressions. Fuzzy logic may be employed to describe a complex chaotic dynamical system. From an application point of view, control is probably the most promising domain of chaos-fuzzy hybrid systems.

Application areas

The following table shows sample potential application areas of chaos in various fields.

Field	Applications
Engineering	Vibration control, stabilization of circuits, chemical reactions, turbines, power grids, lasers, fluidized beds, combustion and many more.
Computers	Switching of packets in computer networks. Encryption. Control of chaos in robotic systems.
Communications	Information compression and storage.Computer network design and management.
Medicine and Biology	Cardiology, heart rhythm (EEG) analysis, prediction and control of irregular heart activity (chaos aware defibrillator)
Management and Finance	Economic forecasting, restructuring, financial analysis and market prediction and intervention.
Consumer Electronics	Washing machines, dishwashers, air conditioners, heaters, mixers.

Further Reading

Strogatz's book is easily understandable and well-written on dynamical systems and chaos. Gleick's book is non-technical but entertaining. Lorenz's and May's are seminal articles cited in this chapter.

K.T. Alligood, T.D. Sauer and J.A. Yorke, *Chaos: An Introduction to Dynamical Systems*, Springer, 2000.

A.B. Çambel, *Applied Chaos Theory: A Paradigm for Complexity*. Academic Press, San Diego, Calif, 1993.

W. Ditto and T. Munakata, "Principles and Applications of Chaotic Systems," *Communications of the ACM*, Vol. 38, No. 11, 96-102, Nov., 1995,.

J. Gleick, *Chaos: Making a New Science*, Viking, 1987.

J. Guckenheimer and P. Holmes, *Nonlinear Oscillations, Dynamical Systems, and Bifurcations of Vector Fields*, Springer,1983, 1990.

A. Lasota and M.C. Mackey, *Chaos, Fractals, and Noise: Stochastic Aspects of Dynamics*, Springer, 1998.

E.N. Lorenz, "Deterministic non-periodic flow," *Journal of the Atmospheric Sciences*, 20, 130-141, 1963.

R.M. May, "Simple Mathematical Models with Very Complicated Dynamics," *Nature*, Vol. 261, 459-467, June, 1976.

E. Ott, *Chaos in Dynamical Systems*, Cambridge University Press, 1993.

E. Ott, T. Sauer, and J.A. Yorke, *Coping with Chaos: Analysis of Chaotic Data and the Exploitation of Chaotic Systems*, Wiley, N.Y., 1994.

T.S. Parker and L.O. Chua, *Practical Numerical Algorithms for Chaotic Systems*, Springer, 1989.

S.H. Strogatz, *Nonlinear Dynamics and Chaos*, Addison-Wesley, 1994.

Journals

There are many journals that carry articles in chaos including the following.

Chaos, American Institute of Physics.

Chaos, Solitons and Fractals, Elsevier Science.

Nonlinearity, Institute of Physics (UK), London Mathematical Society.

International Journal of Bifurcation and Chaos, World Scientific Publishing Co., Singapore.

Index

Activation function, neural networks, 9
Adaptive system, 7
AI (see Artificial intelligence),
Allele, 86, 109
Analysis and prediction, applications of chaotic systems, 243
Aperiodic, chaos, 223
Applications, 162-163, 178, 191
  chaos, 206, 223, 206-245
  fuzzy systems, 156-157
  genetic algorithms, 85-119
  neural networks, 7
  rough sets, 168-171
Approximation space, rough sets, 171, 170-176
Architecture, Boltzmann machine, 70
Architecture, neural networks, 9
Artificial neural network (ANN), 7
Artificial intelligence (AI), 1
Associative memory, neural network, 40
Associativity, fuzzy set, 127
Attractor, 222, 227, 227-233
Attributes, rough sets, 168
Autonomous dynamical system, 210, 214

Backpropagation, neural networks, 7-35
  backward propagation of error corrections, 12
  epoch, 12
  feedforward, 38, 42
  generalized delta rule, neural networks, 17
  hidden layer, 9
  input layer, 9
  learning rate, 18
  momentum rate, 20
  output layer, 10
  target pattern, 12
Backward propagation of error corrections, backpropagation, 12
Basin of attraction, chaos, 228
Bifurcation diagram, chaos, 236
Bifurcation point, chaos, 235
Bifurcation, chaos, 235, 234-238
Binary input, neural network, 39
Binary relation, from a set to a set, 165
Binary relation, ordinary (nonfuzzy), 131
Bipolar input, neural network, 39
Block, partition, 167
Boltzmann machine 69
  architecture, 70
  clamped phase, 75, 77
  clustering, 72
  derivation of delta-weights, 81
  free-running phase, 75, 77
  hidden neuron, 70
  learning process, 73
  negative phase, 75, 77

Boltzmann machine (*cont.*)
network energy, 73
positive phase, 75, 77
probabilities of individual neurons, 74
problem description, 71
self-supervised learning, 69
supervised learning, 69
testing algorithm, 79
unsupervised learning, 69
unsupervised learning algorithm, 76
visible input neuron, 70
visible neuron, machine 70
visible output neuron, 70
Boundary region, rough sets, 172, 183
Bounded solution, chaos, 223

Cartesian product, 130, 122, 163, 165
Certain concept, rough sets, 174
Chaos, 206, 223, 206-245 (see also Dynamical system)
aperiodic, 223
applications, 62-163, 178, 191
attractor, 222, 227, 227-233
basin of attraction, 228
bifurcation, 235, 234-238
bifurcation diagram, 236
chaotic attractor, 233
dynamical systems, 209-223
fractal, 238
logistic map, 235
period-2 cycle, 236
period-4 cycle, 236
period-doubling , 236
strange attractor, 223
Chaotic attractor, 233
Chaotic hybrid system, 243
Chaotic system, 206, 206-245
Chromosomes, genetic algorithms, 85
Clamped phase, Boltzmann machine, 75, 77
Classification,
applications of genetic algorithm, 96
applications of neural networks, 33
Classifier system, genetic algorithms, 118
Clustering, input patterns, neural networks, 63, 72
Cobweb, dynamical systems, 221
Commutativity, fuzzy set, 127
Competitive learning, Kohonen network, 58, 61
Complement, fuzzy set, 126
Composite relation,
relation, fuzzy, 132, 135
ordinary (nonfuzzy), 132
Composition,
fuzzy, 136, 137
ordinary (nonfuzzy), 132
Concentration, fuzzy set, 128
Concepts, rough sets, 169
Condition attributes, rough sets, 168
Confidence factor, rough sets, 174
Connectionist model, neural networks, 7
Content-addressable memory, neural network, 40
Continuous dynamical system, 212
Continuous input, neural network, 39
Continuous membership function, fuzzy set, 124, 145
Control and stabilization, applications of chaotic systems, 242
Control rules, fuzzy, 143
Control
fuzzy, 143-156
applications of genetic algorithm, 96
applications of neural networks, 33
Core, rough sets, 187
Correct solution, genetic algorithm, 87
Crisp set (see Ordinary set), 121
Crossing site, genetic algorithm, 89
Crossover breeding, genetic algorithm, 89

Darwinian evolution, genetic algorithms, 87
Data compression, Kohonen network, 83
Data mining, 162
de Morgan's laws, fuzzy set, 127
Decision attributes, rough sets, 168
Decision table, rough sets, 168
Decisions, rough sets, 168
Definable set, rough sets, 171

Definable, rough sets, 174
Defining length, schema, genetic algorithms, 109
Defuzzification, 150
Degree of membership, fuzzy-set element, 123
Dempster-Shafer theory, 203
Dependency, rough sets, 184
Dependent set with respect to decision attribute, rough sets, 186
Dependent set, rough sets, 186
Dependent variable, dynamical systems, 210
Derivation of delta-weights, Boltzmann machine, 81
Difference equation, 210
Dilation, fuzzy set, 129
Discrete dynamical system, 210
Discrete membership function,
  fuzzy set, 124, 145
  fuzzy variable, 145
Discretization, rough sets, 189
Discriminant analysis, 203
Discriminant index, rough sets, 185
Distributive laws, fuzzy set, 1247
Dynamical system, 210, 210-218 (see also Chaos)
  attractor, 222, 227, 227-233
  autonomous, 210, 214
  basin of attraction, 228
  chaotic attractor, 233
  cobweb, 221
  diffrence equation, 210
  equilibrium solution, 221, 222, 226
  fixed point, 222, 226
  fixed-point attractor, 227
  flow, 219
  linearity, 211, 213
  map, 210
  nonautonomous, 210, 214
  nonlinearity, 211, 213
  orbit, 219
  periodic attractor, 229
  phase space, 218
  quasi-periodic, 230
  recurrence equation, 210
  repeller, 222
  representation of, 210-218
  saddle point, 228
  solution, 210, 212
  state space, 218
  state variable, 218
  time series, 209
  trajectory, 219

Element, fuzzy set, 125
Elementary set, rough sets, 171
Energy function, Hopfield network, 44
Entity, rough sets, 168, 178
Entropy, 196
Epoch, backpropagation, 12
Equality, fuzzy set, 125
Equilibrium solution, dynamical systems, 221, 222, 223, 226
Equivalence classes, 167, 169
Equivalence relation, 165, 178
Evolution, genetic algorithms, 87
Evolutionary computing, 85
Example, rough sets, 168
Excitatory synapse, neural networks, 43
Expected count, genetic algorithm, 92
Extended fuzzy if-then rules tables, 152
Externally definable, rough sets, 175
Externally undefinable, rough sets, 175

Feedbackward neural network, 38, 42
Feedforward neural network, 38, 42
Feedforward, backpropagation, 12, 38
Firing strength, fuzzy rule, 149
Fitness probability, genetic algorithm, 92
Fitness, genetic algorithm, 88
Fixed point, dynamical systems, 222, 226
Fixed-point attractor, chaos, 227
Flow, dynamical systems, 219
Fractal dimension, 240
Fractal, 238
Free-running phase, Boltzmann machine, 75, 77
Fully-interconnected neural network, 42
Fundamental theorem of genetic algorithms, 113
Fuzzification, 149

Fuzzy binary relation (see Fuzzy relation), 130-138
Fuzzy composite relation, 132, 135
Fuzzy composition, 136, 137
Fuzzy control, 143-156
  control rules, 143
  defuzzification, 150
  extended fuzzy if-then rules tables, 152
  firing strength, fuzzy rules, 149
  fuzzification, 149
  fuzzy if-then rule table, 146
  fuzzy inference, 140-143, 149
  fuzzy variable, 143
  input variable, 143
  output variable, 143
  weight, fuzzy rules, 149
Fuzzy graph, 134
Fuzzy if-then rule table, 146
Fuzzy implication, 139
Fuzzy inference, 140-143, 149
Fuzzy logic, 139-143
  applications
  composition, 132, 137
  implication, 139
  induction, 137
  inference, 137-139, 144
Fuzzy relation defined on ordinary sets, 133-138
Fuzzy relation matrix, 134
Fuzzy relation, 130-138
  composite relation, 132
  composition, 132
  fuzzy relation defined on ordinary sets, 133-138
  fuzzy relations derived from fuzzy sets, 138
  fuzzy graph, 134
  max-min product, 135
  relation matrix, 134
Fuzzy relations derived from fuzzy sets, 138
Fuzzy set, 123-130
  associativity, 127
  commutativity, 127
  complement, 126
  concentration, 128
  de Morgan's laws, 127
  degree of membership, 123
  dilation, 129
  distributive laws, 127
  element, fuzzy set, 125
  equality, fuzzy set, 125
  fuzzy singleton, 125
  idempotent, 127
  identity, fuzzy set, 128
  intersection, fuzzy set, 126
  involution (double complement), fuzzy set, 127
  membership function, 123, 124
  membership value, 124
  normal (normalized) fuzzy set, 129
  normalization, fuzzy set, 129
  singleton, fuzzy set, 125
  subset, fuzzy set, 125
  support, fuzzy set, 125
  union, fuzzy set, 126
Fuzzy singleton, 125
Fuzzy systems, 156-157
Fuzzy systems, hybrid, 156-157
Fuzzy variable, 143

Gain ratio, ID3, 197
Gain, ID3, 197
Gene, genetic algorithms, 86, 109
General solution, dynamical systems, 210, 212
Generalized delta rule, neural networks, 17
Genetic algorithm, 85-119
  allele, 86, 109
  application, 102
  classifier system, 118
  correct solution, 87
  crossover breeding, 89
  crossing site, 89
  defining length, schema, 109
  expected count, 92
  fitness, 88
  fitness probability, 92
  fundamental theorem, 113
  gene, 86, 109
  genetic programming, 116-117
  index list, 104
  index table, 104

input-to-output mapping, 95
mating pool, 89
mutation, 86, 89
normalized fitness, 92
order, schema, 109
partially matched crossover (PMX) operation, 107
population, 88
premature convergence, 107
random mutation, 89
recombination breeding, 89
reproduction, 89
schema, 108
schema theorem, 113
similarity template, 108
solution, 87
total fitness, 91
Genetic programming, 116-117
Genotype, genetic algorithms, 86
Global-to-local approach, rough sets, 190
Grade of membership, fuzzy-set element, 123
Graded learning, neural network, 59

Hidden layer, backpropagation, 9
Hidden neuron, Boltzmann machine, 70
Hopfield network, 41-58
energy function, 44
Lyapunov function, 44
Hopfield-Tank model, 46-49
basic equations, 46
iteration process, 47
n-queen, 49
neural networks, 45-55
one-dimensional layout, 46
traveling salesman probem, 55
two-dimensional layout, 48

ID3, 195-202
entropy, information theory, 196
gain, 197
gain ratio, 197
indetermination, probability theory, 196
measure of uncertainty, probability theory, 196
outcome space, probability theory, 195
random variable, probability theory, 195
split info, 196
Idempotent, fuzzy set, 127
Identity, fuzzy set, 128
Implication,
fuzzy, 139
ordinary, 139
Inconsistent information table, rough sets, 170
Independent set with respect to decision attribute, rough sets, 186
Independent set, rough sets, 186
Independent variable, dynamical systems, 210
Indetermination, probability theory, 196
Index list, genetic algorithm, 104
Index table, genetic algorithm, 104
Indiscernibility relation, rough sets, 171
Indispensable with respect to decision attribute, rough sets, 188
Indispensable, rough sets, 187
Indistinguishable element, rough sets, 171
Indistinguishable, rough sets, 171
Induction,
fuzzy set, 137
partition, 167
Information table, rough sets, 168
Inhibitory synapse, neural networks, 43
Input layer, backpropagation, 9
Input variable, fuzzy control, 143
Input-to-output mapping, 95
Internally definable, rough sets, 175
Internally undefinable, rough sets, 175
Intersection, fuzzy set, 126
Involution (double complement), fuzzy set, 127
Isolated solution, chaos, 228

Knowledge representation system, rough sets, 176
Kohonen layer, neural networks, 59
Kohonen (neuron network) model, 59
clustering of input patterns, 63

Kohonen model (*cont.*)
competitive learning, 58, 61
data compression, 83
Kohonen neuron, 59
Kohonen layer, 59
learning vector quantization, 63
radius, 60

Layered neural network, (see Multi-layered neural network), 9, 37
Learning process, Boltzmann machine, 73
Learning rate, backpropagation, 18
Learning vector quantization, Kohonen network, 63
Limit cycle, chaos, 228
Linear dynamical system, 154, 156
Linearity, 211, 213
Linearly inseparable function, 30
Linearly separable function, 30
Local minimum, neural networks, 27
Local-to-global approach, rough sets, 190
Logistic equation, chaos, 235
Logistic map, chaos, 235
Lower approximation, rough sets, 171
Lyapunov function, Hopfield network, 44

Machine learning, applications of genetic algorithm, 95
Map, dynamical systems, 210
Mating pool, genetic algorithm, 89
max-min product, fuzzy relation, 135
Measure of uncertainty, probability theory, 196
Membership function, fuzzy set, 123, 124
Membership value, fuzzy set, 124
Minimal period, dynamical systems, 214
Minimal set, rough sets, 186
Momentum rate, backpropagation, 20
Multilayered neural network, 37
Mutation, genetic algorithm, 86, 89

Negative phase, Boltzmann machine, 75, 77
Negative region, rough sets, 172, 183
Network energy, Boltzmann machine, 73
Neural net, 7
Neural network (NN), 7
activation function, neural networks, 8
applications, 32
architecture, 9
associative memory, 40
backpropagation, 9-38
backward propagation of error corrections, 12
binary input, 39
bipolar input, 39
content-addressable memory, 40
continuos input, 39
epoch, 12
excitatory synapse, 43
feedbackward, 38, 42
feedforward, 12, 38, 42
generalized delta rule, 17
hidden layer, 9
inhibitory synapse, 43
input layer, 10
layered, 9, 37
learning rate, 18
local minimum, 27
momentum rate, 20
multilayered, 9, 37
neuron, 5
nonrecurrent, 38
output layer, 10
perceptron, 28
recurrent, 17, 38, 42
representation , perceptron, 28
sigmoid function, 9, 47, 53
sigmoid function with threshold, 9
supervised learning, 12, 38
synapse, 8, 43
target pattern, 12
transfer function, 8
unsupervised learning, 38
Neuron, 7
Nonautonomous dynamical system, 210, 214
Nonlinear dynamical system, 154, 156

Nonlinearity, 211, 213
Non-recurrent neural network, 38
Normal (normalized) fuzzy set, 129
Normalization, fuzzy set, 129
Normalized fitness, genetic algorithm, 92
n-queen problem, 49

Object, rough sets, 168, 178
Optimization, Hopfield-Tank model, 49-58
Orbit, dynamical systems, 219
Order,
  differential equations, 212
  schema, genetic algorithms, 109
Ordinary set, 121
Outcome space, probability theory, 195
Output layer, backpropagation, 10
Output variable, fuzzy control, 143

Partially matched crossover (PMX) operation, genetic algorithm, 107
Particular solution, dynamical systems, 210, 212
Partition, 167, 168, 169
Pattern classification, applications of genetic algorithm, 96
PDP (Parallel Distributed Processing) model, 7
Perceptron, (simple), 28
Period of a solution, chaos, 228
Period-2 cycle, chaos, 236
Period-4 cycle, chaos, 236
Period-doubling, chaos, 236
Periodic attractor, chaos, 229
Periodic solution, 228
Phase point, dynamical systems, 219
Phase space, dynamical systems, 218
Phenotype, genetic algorithms, 86
Population, genetic algorithm, 88
Positive phase, Boltzmann machine, 75, 77
Positive region, rough sets, 172, 183
Prediction,
  applications of genetic algorithm, 96
  applications of neural networks, 33
Premature convergence, genetic algorithm, 107
Probabilities of individual neurons, Boltzmann machine, 74
Problem description, Boltzmann machine, 71
Product of partitions, 169

Quasi-periodic, chaos, 230

Radius, Kohonen network, 60
Random mutation, genetic algorithm, 89
Random variable, probability theory, 195
Recombination breeding, genetic algorithm, 89
Recurrence equation, 210
Recurrence relation, 210
Recurrent neural network, 17, 38, 42
Reduct, rough sets, 186
Reinforcement learning, neural network, 59
Relation matrix, fuzzy, 134
Relation, from a set to a set, 165
Relation, fuzzy, 130-138
Relation, ordinary (nonfuzzy), 131
Relative core, rough sets, 188
Relative reduct, rough sets, 186
Repeller, dynamical systems, 222
Representation, perceptron, 28
Reproduction, genetic algorithm, 89
Rough set, 175, 162-204
  applications, 162-163, 178, 191
  approximation space, 171
  attributes, 168
  boundary region, 172, 183
  certain concept, 174
  concepts, 169
  condition attributes, 168
  confidence factor, 174
  core, 187
  decision attributes, 168
  decisions, 168
  definable, 174
  dependency, 184
  dependent set, 186
  dependent set with respect to decision attribute, 186
  discretization, 189
  discriminant index, 185

Rough set (*cont.*)
elementary set, 171
entity, 168
equivalence classes, 167, 169
example, 168
inconsistent information table, 170
independent set, 186
independent set with respect to decision attribute, 186
information table, 168
indispensable, 187
indispensable with respect to decision attribute, 188
indistinguishable element, 171
indiscernibility relation, 171
knowledge representation system, 176
lower approximation, 171
minimal set, 186
negative region, 172, 183
objects, 168
partition, 167, 169
positive region, 172, 183
reduct, 186
relative core, 188
relative reduct, 186
significance, 185
uncertain concept, 174
universe, 168
upper approximation, 171
Roughly definable, 175
Roughly dependent, 184

Saddle point, chaos, 228
Schema theorem, genetic algorithms, 113
Schema, genetic algorithms, 108
Self-organization, neural network, 59
Self-supervised learning, Boltzmann machine, 69
Sigmoid function with threshold, neural-network activation function, 9
Sigmoid function, neural-network activation function, 9, 47, 53
Significance, rough sets, 185
Similarity dimension, 241
Similarity template, genetic algorithms, 108
Simulated annealing 63
algorithm, 65
example, TSP, 66
Singleton, fuzzy set, 125
Soft computing, 5
Solution,
dynamical systems, 210, 212
genetic algorithm, 87
Split info, ID3, 197
Stable solution, dynamical systems, 222
State point, dynamical systems, 219
State space, dynamical systems, 218
State variable, dynamical systems, 218
Steady state, dynamical systems, 210, 212
Steepest descent method, 18
Strange attractor, chaos, 223
Subset, fuzzy set, 125
Supervised learning, neural networks, 38
Supervised learning, Boltzmann machine, 69
Supervised learning, neural networks, 12, 38
Support, fuzzy set, 125
Symbolic AI, 2
Symbols for
fuzzy systems, 156-157
rough sets, 168-171
Synapse, neural network, 8, 43
Synthesis, applications of chaotic systems, 242

Target pattern, backpropagation, 12
Testing algorithm, Boltzmann machine, 79
Time series, dynamical systems, 209
Time waveform, dynamical systems, 209
Time-periodic, dynamical systems, 214
Torus, chaos, 230
Total fitness, genetic algorithm, 91
Totally dependent, rough sets, 184
Totally independent, rough sets, 184
Totally non-definable, rough sets, 175
Totally undefinable, rough sets, 175

Trajectory, dynamical systems, 219
Transfer function, neural networks, 8
Transient state, dynamical systems, 210, 212
Traveling salesman problem (TSP),
  application of genetic algorithm, 102
  application of the Hopfield-Tank model, 55
  simulated annealing, 66
Triangular membership function, fuzzy variable, 145

Uncertain concept, rough sets, 174
Undefinable, rough sets, 175
Union, fuzzy set, 126
Universe, rough sets, 168
Unstable solution, dynamical systems, 222
Unsupervised learning algorithm, Boltzmann machine 76
Unsupervised learning, 38, 59
Unsupervised learning, Boltzmann machine 69
Upper approximation, rough sets, 171

Visible input neuron, Boltzmann machine, 70
Visible neuron, Boltzmann machine 70
Visible output neuron, Boltzmann machine, 70
von Koch curve, 238

Waveform, dynamical systems, 209
Weight, fuzzy rule, 149

TEXTS IN COMPUTER SCIENCE *(continued from page ii)*

Kizza, Ethical and Social Issues in the Information Age, Second Edition

Kozen, Automata and Computability

Kozen, Theory of Computation

Li and Vitányi, An Introduction to Kolmogorov Complexity and Its Applications, Second Edition

Merritt and Stix, Migrating from Pascal to C++

Munakata, Fundamentals of the New Artificial Intelligence: Neural, Evolutionary, Fuzzy and More, Second Edition

Nerode and Shore, Logic for Applications, Second Edition

Pearce, Programming and Meta-Programming in Scheme

Revesz, Introduction to Constraint Databases

Schneider, On Concurrent Programming

Skiena and Revilla, Programming Challenges: The Programming Context Training Manual

Smith, A Recursive Introduction to the Theory of Computation

Socher-Ambrosius and Johann, Deduction Systems

Stirling, Modal and Temporal Properties of Processes

Zeigler, Objects and Systems